by JACK W. CRENSHAW

All About Fourier Analysis

The first installment in a new series exploring the world of time, frequency, and waveforms.

Is there an engineer alive who hasn't heard the terms Fourier series, Fourier transform, and Fourier analysis? Probably not. Even if you haven't used the techniques in quite a while, if ever, you probably remember that it has something to do with sine waves. But that simple statement seems entirely too innocuous to indicate the profound effect that Fourier analysis has had on almost every aspect of engineering, and on everyday life.

In this two-part series, we'll be looking at the motivation and concepts behind Fourier analysis, and why it's so important. This month, we'll look at the Fourier series, which allows us to represent a periodic function by a sum of many different sine waves. In the process, I'll give you a neat and simple, but powerful, little algorithm of mine, which I call the "half-fast Fourier transform." Next time, we'll extend these concepts of sums and harmonics to the realm of infinities and infinitesimals, which will lead us to the all-important Fourier transform and its implementation in the famous Fast Fourier Transform (FFT).

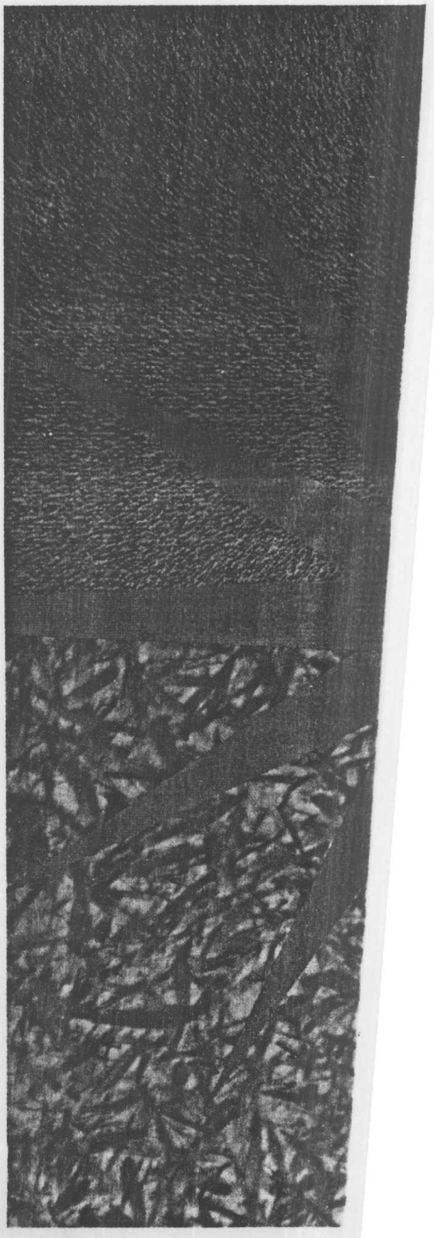
GETTING MOTIVATED

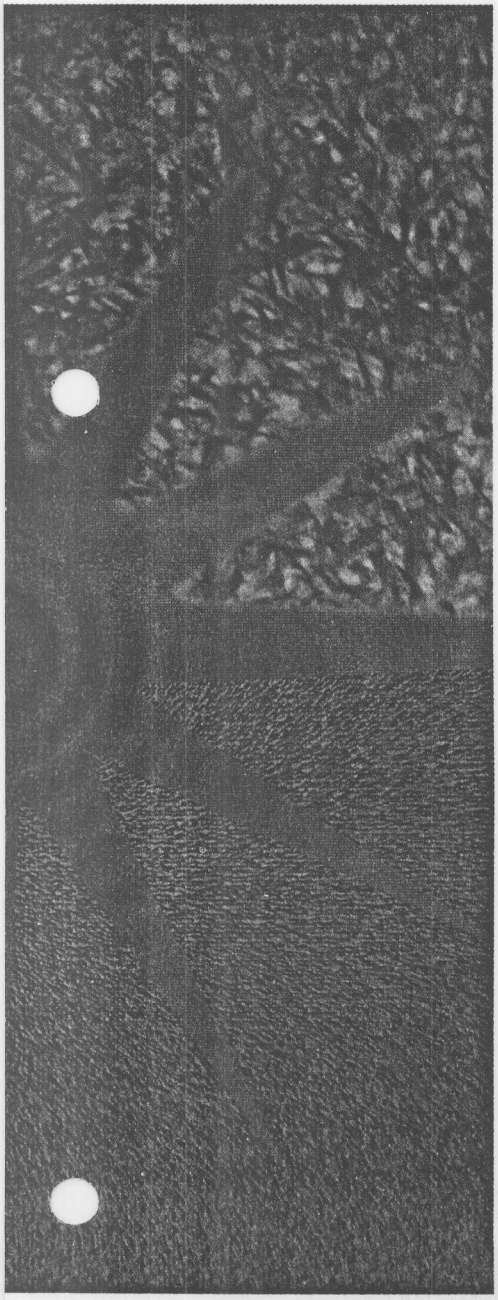
When I was in school, I found that I always got more out of a class if I knew what direction the professor's lectures were taking, and why. Whenever I set out to explain a new technique, I try to provide the reasoning behind it, so readers can understand the background and usefulness of the technique. Basically, I like to understand and explain what might have been going through the head of the inventor.

Providing the reasoning for some topics is easier than others. Few have such a wealth of background as Fourier analysis; in this case, we have an embarrassment of riches. Though you may not realize it, the concepts we need for the subject of Fourier analysis are already as familiar to you as your thumb and forefinger.

When you were shopping for your first stereo system, you no doubt already knew that if an audio system is to faithfully reproduce music or other sounds, it must have "flat" frequency response. That is, it must amplify all frequencies in the audio range (20 Hz to 20,000 Hz) with equal amplitude. If it doesn't, the music won't sound right. You also knew that every amplifier has a pair of tone control knobs labeled "bass" and "treble." Turning the bass up or down affects the reproduction of low frequencies, while the treble knob affects high frequencies. Turn the treble up and the bass down, and the music sounds thin and sibilant. Turn them the opposite way and the music sounds muddy and muffled with booming bass drums.

When you first tweaked one of those knobs yourself, you may have been surprised to discover that these knobs made a profound difference, even if only one instrument, say a violin, were playing. That's because all musical instruments generate overtones, or harmonics, which are sounds with frequencies above the fundamental tone that defines the note being played. Adjusting the tone controls affects the





sub aqua design

Mathematically, the Fourier series is only a subset of a larger set of methods based on orthogonal functions.

relative amplitudes between the fundamental and its overtones, which changes the sound quality.

Implicit in all this talk about tones, fundamentals, and overtones is the assumption that a repetitive sound can be decomposed into a series of sine waves of different frequencies. That assumption, originally due to Daniel Bernoulli, is the basis of all Fourier analysis: to approximate an arbitrary, periodic function with a series of sine (or cosine) waves. Why would we want to do that? Mainly, because we know a lot about how to manipulate sine waves—we can easily take their derivatives and integrals, apply trig identities to them, and so on. We probably can't do that with functions of arbitrary complexity. The Fourier series is yet another example of an approach I've stressed in these pages many times before: If you can't solve the problem you have, turn it into one that you can.

Once we have a signal in its Fourier representation, we can also do tricks with it—the mathematical equivalent of tone controls. We can emphasize the high or low frequencies, filter out unwanted frequencies, and so on. Tone controls do that using analog circuitry, and we can do the same thing digitally, using a computer or digital signal processor. Done properly, the digital approach can be much more effective than any analog circuit, and is the foundation of digital signal processing.

When J.B. Fourier first published his technique in 1822, the concept must have seemed wonderful and mysterious and perhaps even a bit risky to his colleagues. But thanks to that ubiquitous stereo system, to us it seems as natural as breathing. You know the method works because you can hear it with your ears.

ORTHOGONAL FUNCTIONS

Mathematically, the Fourier series is only a subset of a larger set of methods based on orthogonal functions. Such functions have many practical uses far beyond those of the Fourier series. Furthermore, we'll find that most of the concepts are actually easier to understand in the broader view, so we'll take the time to explore the properties of orthogonal functions in general before specializing to the Fourier series.

Suppose we have an arbitrary function $\phi(x)$. We'd like to approximate this function in terms of other (presumably simpler or more tractable) func-

All About Fourier Analysis

tions $\phi_n(x)$. The subscript n is there because we must have more than one such function, or else $\phi(x)$ would have to be f(x). We can write the approximation as a linear combination of the base functions:

$$f(x) = a_0\phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) + \cdots + a_n\phi_n(x) + \cdots \quad (1)$$

(The trailing ellipsis is meant to indicate that the series might be infinite.) Expressed as a summation, Equation 1 becomes:

$$f(x) = \sum_{n=0}^{\infty} a_n\phi_n(x) \quad (2)$$

Of course, we've seen plenty of series before, usually in the guise of power series. You can now see that the form in Equation 2 is much more general; a power series is merely a special case of this more general form, where:

$$\phi_n(x) = x^n \quad (3)$$

Our next challenge is to discover the right values to plug in for those coefficients a_n. At first glance, this may seem an impossible task, but as you will see, it can be done even with the most general assumptions about the functions $f_n(x)$.

First, we must recognize that we are extremely unlikely to be able to approximate f(x) over an infinite range. Much more likely, we must look at it only over some finite range, say r to s. Now let's do something completely crazy. With no justification at all, we'll multiply f(x) by one of our functions, $\phi_m(x)$. Then we'll integrate over the range to get:

$$\int_r^s \phi_m(x)f(x)dx = \int_r^s \left[\sum_{n=0}^{\infty} a_n\phi_m(x)\phi_n(x)\right]dx$$

We can interchange the order of summation and integration, which amounts to integrating term by term:

$$\int_r^s \phi_m(x)f(x)dx = \sum_{n=0}^{\infty} a_n\left[\int_r^s \phi_m(x)\phi_n(x)dx\right] \quad (4)$$

Well! It certainly doesn't seem that we've made our lives any easier—this is a messy-looking equation, containing not just one, but both of those dread symbols, the integration and summation symbols. But take another look. The integral term in brackets on the right doesn't contain f(x), only the functions $\phi_n(x)$, which are known and unchanging. So once we define our function set (and the range limits, r and s), the whole term becomes a simple constant, computable beforehand. Let's call it p_{mn}. Obviously, we have more than one such constant—in fact, a potential infinity of them, one for each combination of m and n. But conceptually, we can organize them into a matrix Also, we can define a vector u such that:

$$u_n = \int_r^s \phi_n(x)f(x)dx \quad (5)$$

The elements of this vector, of course, do contain f(x), so we must perform the integrals for each case, but they are straightforward enough.

We can now write Equation 4 in its equivalent matrix form:

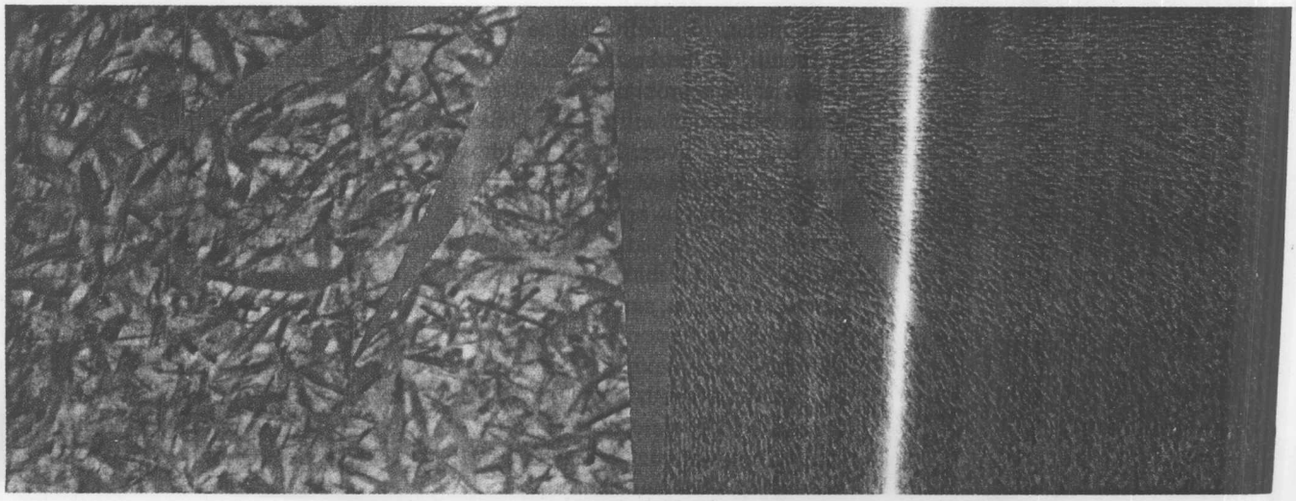
$$\mathbf{u} = \mathbf{P}\mathbf{a} \quad (6)$$

Ignore for the moment some embarrassing technical details, such as the fact that we chose our indexing in Equation 4 to run from zero instead of one and that **a**, **u**, and **P** are all potentially infinite in size. As my major professor used to say, "conceptually there's no problem."

Conceptually, then, we can invert Equation 6 to find the desired coefficients, as shown:

$$\mathbf{a} = \mathbf{P}^{-1}\mathbf{u} \quad (7)$$

As you can see, we really didn't com-



All About Fourier Analysis

When we evaluate the integrals at their limits, all the sine functions cancel and we're left with zero.

plicate the problem by multiplying and integrating, we solved it. Conceptually, that is. You might still be wondering, however, whatever possessed us to take such a seemingly bizarre step. The short but somewhat unsatisfying answer is, "it works." The best answer I can give you is that we must somehow get into the solution, the behavior of f(x), not only at specific points but over the entire range of interest. An integral is the only way to do that. As for multiplying by $\phi_m(x)$, we must also somehow isolate the effect of one of the coefficients from the total. Because squaring a function is sure to emphasize its effect, the product seems to draw the influence of that one function out of the noise. The trick will become much clearer in the next step.

You see, so far we haven't said anything about the orthogonality of the functions; in fact, we haven't even defined the term. But look again at the form of the matrix P. Since:

$$p_{mn} = \int_r^s \phi_m(x)\phi_n(x)dx \qquad (8)$$

the matrix is symmetric—$p_{mn} = p_{nm}$. Setting m = n gives us the diagonal terms:

$$p_{nn} = \int_r^s [\phi_n(x)]^2 dx \qquad (9)$$

Because the function is squared, these terms must all be positive, and can be zero only if $\phi_n(x)$ is zero throughout its range (a pretty unlikely event). But what about all those off-diagonal terms? Wouldn't it be nice if, by some miracle, they all turned out to be zero? In that case, the matrix would be diagonal, and its inversion would be trivial. In the summation of Equation 4, only one term would be non-zero, so the equation would become, simply:

$$\int_r^s \phi_n(x)f(x)dx = a_n p_{nn}$$

or:

$$a_n = \frac{u_n}{p_{nn}} \qquad (10)$$

where u_n is defined as in Equation 5. Our seemingly impossible task has suddenly gotten very simple indeed. Unless we're awfully silly in choosing the functions $\phi_n(x)$, the integrals defining p_{nn} will be available analytically, so we need only compute the values of u_n to get our desired coefficients.

A set of functions $\phi_n(x)$ having the property that the integrals of their products is zero, for all m and n except m=n, is called an orthogonal set. There still remains the question of whether we're asking the impossible in seeking such a set. After all, we're talking an infinite number of such functions; what are the chances that the integral be zero for all those infinite products? Fortunately, we can easily show that at least one set of such functions exists: the sine and cosine functions. Consider the integral:

$$I = \int_{-\pi}^{\pi} \sin mx \sin nx dx \qquad (11)$$

A trig identity gives:

$$\sin a \sin b = \tfrac{1}{2}[\cos(a-b) - \cos(a+b)]$$

Thus:

$$I = \tfrac{1}{2}\int_{-\pi}^{\pi} [\cos(n-m)x - \cos(n+m)x]dx \qquad (12)$$

The integrand is now simply the difference of two cosine functions, and the integral is straightforward:

$$I = \tfrac{1}{2(n-m)}[\sin(n-m)x]_{-\pi}^{\pi} - \tfrac{1}{2(n+m)}[\sin(n+m)x]_{-\pi}^{\pi}$$

Now, the sine function is an odd function. Not odd as in peculiar, but odd in the sense that:

```
sin (-x) = -sin x
```

When we evaluate the integrals at their limits, all the sine functions cancel and we're left with zero, as we'd hoped, for all cases except the one where m = n. That case must be treated separately:

$$I = \int_{-\pi}^{+\pi} \sin^2 nxdx = \int_{-\pi}^{+\pi} \tfrac{1}{2}(1 - \cos 2nx)dx$$

$$= \tfrac{1}{2}x\Big|_{-\pi}^{+\pi} = \frac{2\pi}{2} = \pi \qquad (13)$$

We get similar results for the case of cosine functions and a combination of one sine and one cosine function. We have one more special case to consider, and that's the one for cos nx, where n = 0. This, of course, yields the constant "function" 1. The integral of the product of this function with the sine function is, of course:

$$I = \int_{-\pi}^{+\pi} \sin nxdx$$

which is clearly zero and the same for the cosine function. There remains only the integral of the function multiplied by itself, which is, of course:

$$I = \int_{-\pi}^{+\pi} dx = \pi - (-\pi) = 2\pi \qquad (14)$$

So what does all this tell us? Simply that it's really possible to define a set of orthogonal functions because the set consisting of 1, sin nx, and cos nx is

All About Fourier Analysis

just such a set, on the interval $-\pi..\pi$. This set of functions is, of course, the basis for the Fourier series. Equations 13 and 14 also give us the values of p_{nn} for this function set.

A DIGRESSION

We've seen that sine and cosine waves form an orthogonal set. Clearly, the terms in a power series are not orthogonal functions, since:

$$\int x^{n+m} dx$$

will not be zero over any interval for all m and n. However, take a look at the first two functions in such a series:

$$T_0(x) = 1$$

$$T_1(x) = x$$

These two functions are, in fact, orthogonal. If we evaluate the cross integral:

$$I = \int_{-1}^{1} T_0(x) T_1(x) dx$$

we get:

$$I = x\Big|_{-1}^{+1} = 1 - 1 = 0$$

At least we have two polynomial functions that are mutually orthogonal over a specific interval. Can we find a third? We know that the term x_2 by itself won't work, but what about a different quadratic in x? Yes, the function:

$$T_2(x) = 2x^2 - 1 \qquad (15)$$

is orthogonal to both other functions over the specified interval. Sharp-eyed readers will spot the three functions as the first three Chebyshev polynomials, of which there is an infinite number. These polynomials do indeed compose another infinite set of orthogonal functions. Not surprisingly, many other sets have also been defined, each best suited for dealing with a particular kind of problem.

BACK TO FOURIER

I've already told the punch line of my story by giving you the orthogonal functions used in the Fourier series—they are the sine and cosine functions. Specifically, if a given function is periodic with period 2π, the Fourier series corresponding to the function is:

$$f(x) = \frac{a_0}{2} + a_1 \cos x + a_2 \cos 2x + \\ + a_3 \cos 3x + b_1 \sin x + \\ + b_2 \sin 2x + b_3 \sin 3x + \cdots$$

or, more concisely:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx)$$

The first, constant coefficient is usually separated out and given the factor of one-half shown, so we may use a single definition for all the coefficients:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \cos nx dx \quad (n = 0, 1, 2...) \qquad (17)$$

and:

$$b_n = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \sin nx dx \quad (n = 1, 2, 3...) \qquad (18)$$

The multipliers of these integrals come from the definition of p_{nn} in Equation 9, and our integrals obtained in Equations 13 and 14. The extra factor of 1/2 in Equation 14 is the reason for the factor of two in the a_0 term. We have no b_0 term because, of course, $\sin(0) = 0$, so the term is zero regardless of the value of the coefficient.

One of the great advantages of the Fourier series is that it can handle functions that are poorly behaved. Specifically, it's valid for functions that have discontinuities in them. This is very important to mathematicians because most math techniques such as differentiation require functions to be continuous. Using the Fourier series, we replace a function with discontinuities by a summation of an infinite number of functions, each of which is itself continuous. That's a neat mathematical hat trick.

When Fourier first introduced his series, there was much concern over when and if we could be sure the series converged. Fortunately, it was later shown by Dirichlet that it does converge as long as certain rather loose conditions are met. These are the Dirichlet conditions:

- The function must have at most a finite number of discontinuities.
- It must have a finite number of maxima and minima.
- These maxima and minima must be bounded.

These conditions are easily met for all but pathological functions, so for all practical applications, we can be assured that the series will indeed converge. Also, any integrals or derivatives of the series will converge to the integrals or derivatives of f(x). At discontinuities, the series converges to the midpoint of the extreme values. These characteristics delighted the mathematicians because they let them tame otherwise unpleasant functions.

SYMMETRIES

We can sometimes make use of symmetry properties to simplify the Fourier series for a function. We can split the integrals of Equation 17 and 18 into two parts like this:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{0} f(x) \cos nx dx + \\ \frac{1}{\pi} \int_{0}^{\pi} f(x) \cos nx dx$$

$$= \frac{1}{\pi} \int_{0}^{\pi} f(-x) \cos nx dx + \\ \frac{1}{\pi} \int_{0}^{\pi} f(x) \cos nx dx$$

All About Fourier Analysis

or:

$$a_n = \frac{1}{\pi}\int_0^{\pi}[f(x) + f(-x)]\cos nx dx \quad (19)$$

Similarly:

$$b_n = \frac{1}{\pi}\int_0^{\pi}[f(x) - f(-x)]\sin nx dx \quad (20)$$

An even function is one that is symmetric about the x = 0 point, as shown in Figure 1a. For such a function:

$$f(-x)=f(x) \quad (21)$$

and the coefficients become:

$$a_n = \frac{2}{\pi}\int_0^{\pi} f(x)\cos nx dx$$

(even functions only) (22)

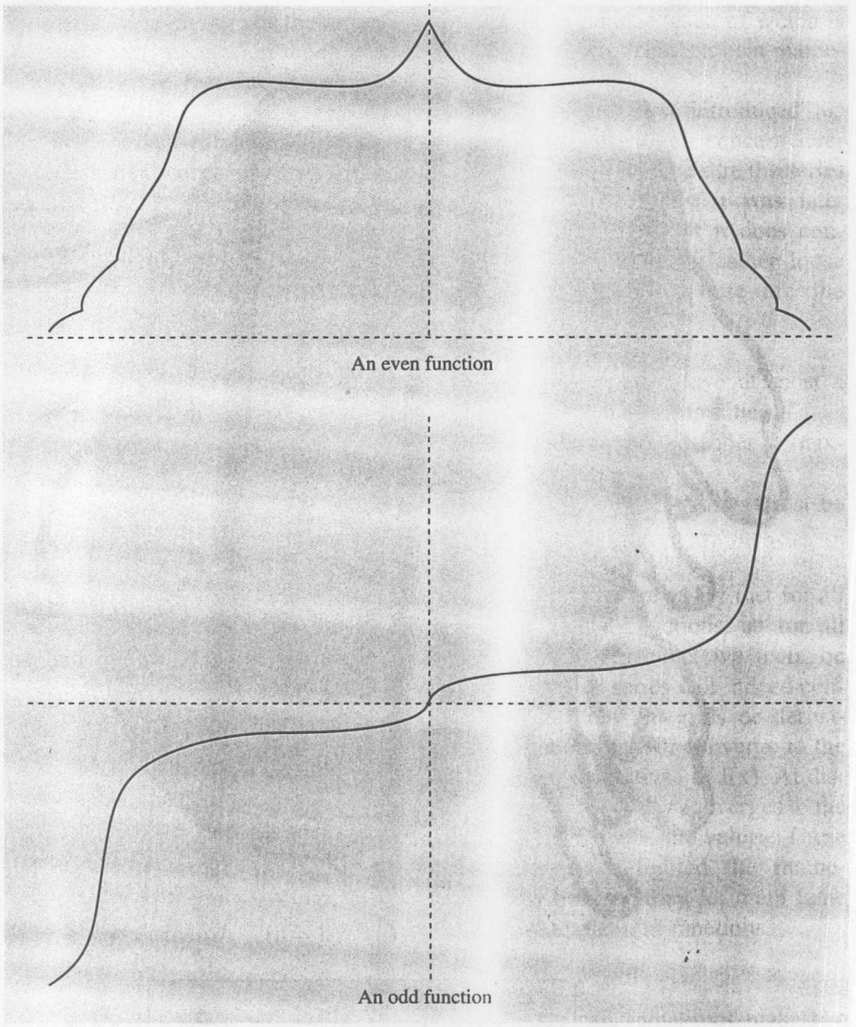
$$b_n = 0$$

Similarly, a function is odd or antisymmetric if its left side is a negative version of the right one, as shown in Figure 1b. For this case:

$$f(-x) = -f(x) \quad (23)$$

For odd functions, the equations for the coefficients become:

FIGURE 1
Even and odd.



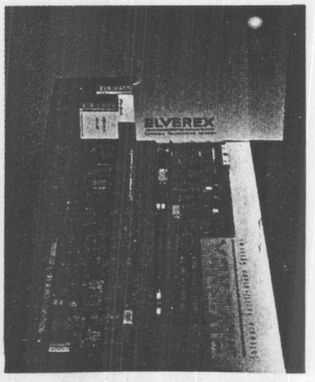
EVALUATOR THE ONLY SOFTWARE TESTING SOLU ON YOU CAN TRUST

Before You Invest in a Software Based Test Product, Ask Those Vendors if They Can Do This:

- Support for WINDOWS NT, WINDOWS, OS/2, DOS, UNIX, and other environments with one product
- Support for any GUI application with custom controls
- Non-intrusive benchmarking of user interactive applications
- Non-intrusive compatibility testing
- Verification of actual graphic data as the user sees it without having to simulate or replace parts of the OS.
- Use Microsoft or Borland C compatible high level test scripts
- Do Systems level testing including bus I/O, Serial I/O, and analog I/O monitoring.

We Support All This and More Now! Call Today for Information on a Testing Solution that Meets Your Needs.

NT NON STANDARD CONTROLS DOS GRAPHICS UNIX



Eastern Systems Inc.
The Software Testing Solution
P.O. Box 1087, Westboro, MA 01581-6087
(508) 366-3223 FAX (508) 366-1520

SERVICE CARD

$$a_n = 0$$

(odd functions only) (24)

$$b_n = \frac{2}{\pi}\int_0^{\pi} f(x)\sin nx dx$$

Either way, half the coefficients vanish. This simplification works in our favor in two ways. Not only do we reduce the number of coefficients to compute, but also, when we evaluate the series, we can skip the computation of either all the sines or all the cosines. Needless to say, it pays to look for such symmetries. Bear in mind, also, that functions may be symmetric about points other than x = 0. We can still get the advantages of symmetries, though, by a change of variables.

APPROXIMATIONS

Theoretically, the Fourier series for a function doesn't just approximate that function, it is the function, exactly. In practice, we never really get a perfect fit to the function because we can't compute an infinite number of coefficients or add up an infinite number of sine waves of increasingly higher frequency. Fortunately, we don't have to. Despite the aspirations of the more rabid audiophiles, an amplifier doesn't really have to pass frequencies somewhere above gamma rays to sound good. A frequency response beyond the limit of audibility is sufficient. In the same way, in many applications of Fourier series we can get by with approximating a function by its Fourier series expansion, truncated to some finite number of terms.

At this point, most textbooks offer some examples to show how well (or poorly) a Fourier series fits a given function. I won't break with tradition; let's compute the series for a few sample inputs. If we choose simple functions, we can get the integrals in closed form. Consider, first, the ubiquitous square wave, shown as an odd function in Figure 2. For this function:

$$f(x) = 1 \qquad (0 \le x < \pi)$$

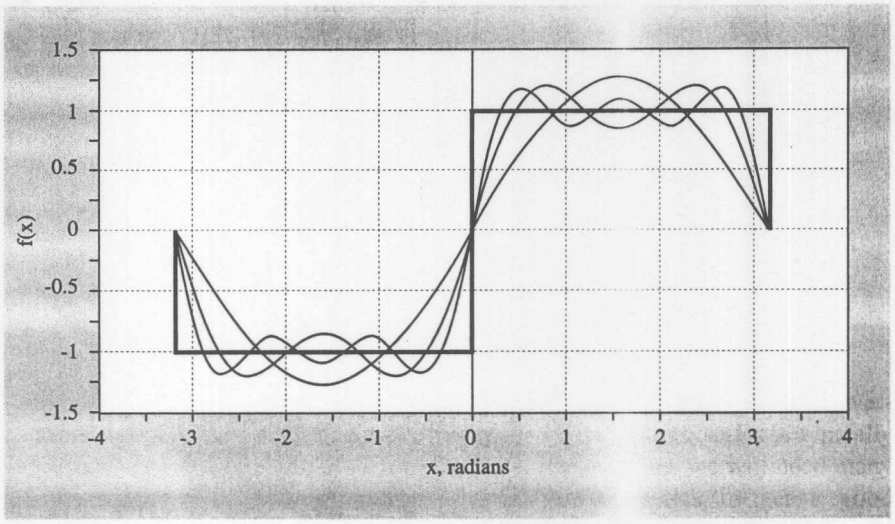
$$f(x) = -1 \qquad (-\pi \le x < 0) \qquad (25)$$

Since the function is odd, Equation 24 applies. The integrals become:

$$b_n = \frac{2}{\pi}\int_0^{\pi} \sin nx dx$$

Integrating gives us:

FIGURE 2
Square wave.



SEE US AT ESC BOOTH 319



(actual size)

GIVE US AN INCH, AND WE'LL GIVE YOU THE UNIVERSE.™

Universe™ from Newbridge Microsystems. It's the first and only high performance single chip PCI to VMEbus interface on the market today. Snap it on, and you've got a ready-made solution — all in a ball grid array package that takes up *oh so little* space on your board.

For more information call
1-800-267-7231
or (613) 592-0714

NEWBRIDGE MICROSYSTEMS
THE POWER WITHIN

Newbridge Microsystems • 603 March Road • Kanata • ON K2K 2M5 Canada

CIRCLE # 48 ON READER SERVICE CARD

$$b_n = \frac{2}{n\pi}\left|-\cos nx\right|_0^{\pi}$$

$$= \frac{2}{n\pi}[-\cos n\pi + \cos 0]$$

which gives:

$$b_n = \frac{2}{n\pi}(1 - \cos n\pi) \qquad (26)$$

Now, cos nπ has a value of 1, if n is an even number, and -1, if it's odd. So we can write:

$$b_n = \frac{4}{n\pi} \quad \text{(n odd)}$$

$$b_n = 0 \quad \text{(n even)} \qquad (27)$$

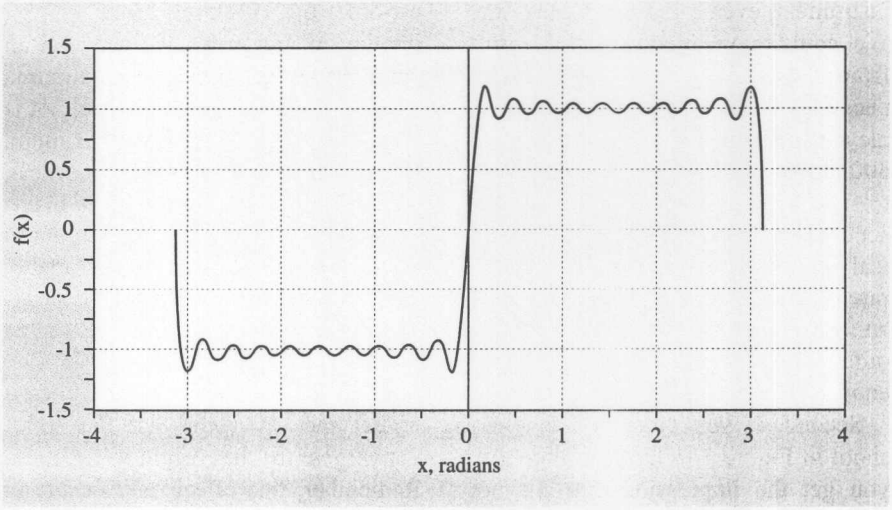
The coefficients for the first few terms are shown in Table 1. The graphs of the first three approximations (orders 1, 3, and 5) are also shown in Figure 2. As you can see, the first approximation is simply a sine wave with amplitude 1.27. As we might expect, the approximation gets better for increasing order. The 19th-order approximation, shown in Figure 3, is beginning to look quite good. This figure gives us a good feel for how the Fourier series handles discontinuities. Sudden, discontinuous jumps in amplitude are replaced by steep but continuous changes, with a little overshoot and ringing nearby. As promised, the value at the discontinuity is the average of the two extreme values. The little ripples in the tops of the wave are also typical of Fourier approximations. Looking at the figure, it's easy to see that the effect of the next higher term would be to smooth out those ripples (and add new, higher-frequency ones).

From the graphs, we can make two important observations. First, Dirichlet's analysis guarantees that the series always converges, but it doesn't promise how fast. In fact, the convergence of the Fourier series is typically slow, which means we will need many terms in the series to get a good approximation. Stated in audio terms,

TABLE 1
Coefficients for square wave

n	b_n
1	1.2732395
3	0.4244132
5	0.2546479
7	0.1818914
9	0.1414711
11	0.115749
13	0.0979415
15	0.0848826
17	0.0748964
19	0.0670126

FIGURE 3
19th order.





SEE US AT ESC BOOTH 319

GIVE US AN OPENING, AND WE'LL TAKE THE MARKET BY STORM.

SCV64™ from Newbridge Microsystems and our partners at Motorola and DY4 Systems Inc. It was the first single chip VMEbus interface controller ever introduced. We saw an opening for a solution that would simplify and speed the board design process. And now our SCV64 is the highest performing solution — and the hottest seller — on the market today.

For more information call
1-800-267-7231
or (613) 592-0714

NEWBRIDGE MICROSYSTEMS
THE POWER WITHIN

Newbridge Microsystems • 603 March Road • Kanata • ON K2K 2M5 Canada

CIRCLE # 83 ON READER SERVICE CARD

we need to reproduce many harmonics to accurately represent the sound of an instrument, even a low-frequency one. We could have predicted this result from Equation 27; the amplitude decreases only as 1/n. To get an accuracy of one part in 1,000, we'd need 500 terms. Obviously, we're not going to get the kind of part-per-million accuracy we're used to in other numerical approximations. Fortunately, we rarely need that kind of accuracy, and an approximation as good as the 19th-order fit of Figure 3 is often good enough.

Second, if you compare the overshoot in Figure 3 to those in Figure 2, you get the impression that it's not really decreasing much—the overshoots for third, fifth, and 19th order seem almost the same. You might well ask, how many terms do I need before the overshoot starts decreasing? The answer: it never will.

Surprisingly, the overshoot has a constant amplitude for all but the lowest-order fits. However, for an approximation of any order, we can find an equation for the maxima and minima with a little calculus. The overshoot is the value of f(x) at the first maximum. If we evaluate this for various orders, we find that the height of this maximum does not converge to zero, but to a value independent of n, and roughly 11% larger than the value of the original function f(x). This effect is called the Gibbs phenomenon.

The phenomenon may seem to contradict the statement that the series converges to the original function. Remember, this effect only occurs in the vicinity of a discontinuity, where all bets are off anyway. At any point away from the discontinuity, even by an infinitesimal distance, the series behaves normally. In practice, the Gibbs phenomenon is a minor annoyance, nothing more, albeit an interesting and surprising one.

Let's try another function, the sawtooth, which is shown in Figure 4. For this function:

$$f(x) = x \tag{28}$$

This is another odd function, so the coefficients are given by:

$$a_n = 0$$
$$b_n = \frac{2}{\pi}\int_0^{\pi} x \sin nx dx \tag{29}$$

A table of integrals gives:

$$\int x \sin x dx = \sin x - x \cos x$$

This equation has the same form as our equation for b_n that has the factor, n, in the sine function. If we let y = nx, Equation 29 becomes:

$$b_n = \frac{2}{n^2\pi}\int_0^{n\pi} y \sin y dy$$

Our formula then gives us:

$$b_n = \frac{2}{n^2\pi}\left|\sin y - y\cos y\right|_0^{n\pi}$$
$$= \frac{2}{n^2\pi}[(\sin n\pi - n\pi\cos n\pi) -$$
$$(\sin 0 - 0\cos 0)]$$
$$= \frac{2}{n^2\pi}(\sin n\pi - n\pi\cos n\pi)$$

Now, sin nπ = 0 for all n, and cos nπ is either -1 or 1. We get, finally:

$$b_n = \frac{2}{n}(-1)^{n+1} \tag{30}$$

Interestingly, the amplitude of th coefficients goes again as 1/n, just a for the square wave. The difference that this series contains all harmonic not just the odd ones, and they alte nate in sign. The first few odd orders the approximation are shown in Figu 4 and the 19th order in Figure 5. Aga note the overshoot (Gibbs phenor non) and ripple that are characteri of the approximation.

All New Features!
PromICE
▼Ethernet Support
▼8051 Debugging
▼XRAY+/pSOS Support
CALL TODAY! 1-800-PROMICE
(1-800-776-6423)
Grammar Engine Inc.
921 Eastwind Dr., Suite 122
Westerville, OH 43081
Phone 614/899-7878
Fax 614/899-7888
CIRCLE # 50 ON READER SERVICE CARD

CHANGE OF VARIABLES

By now, you're probably convinced that the Fourier series really works and provides a good approximation for any periodic function. So far, though, we've only shown functions that are periodic with period 2π and centered at zero. Real functions aren't always so accommodating. In general, a periodic function of time has period T and, equivalently, a frequency given by:

$$f = 1/T \tag{31}$$

(Sorry about the duplicate use of the symbol f, but its use for both the frequency, f, and the function, f(t), are rather traditional. When I refer to the function, I will always do so with the parentheses in place, so there should be no confusion.)

We'll usually find it convenient to specify the function over the range 0..T, rather than centered at zero. So we need a relationship between x and t such that:

when $x = -\pi$, $t = 0$
when $x = \pi$, $t = T$

The transformation:

$$t = \frac{T}{2}\left(1 + \frac{x}{\pi}\right) \tag{32}$$

does the job. Reverting this relationship gives:

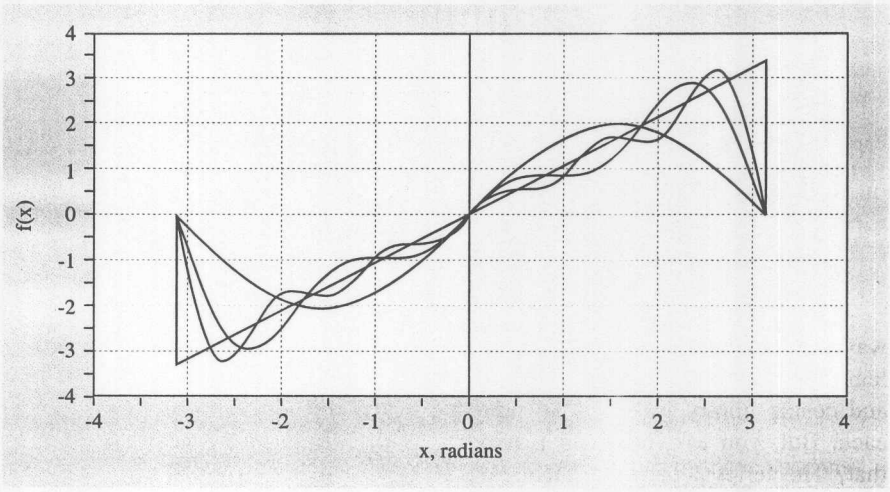
$$x = \pi\left(\frac{2t}{T} - 1\right) \tag{33}$$

Many texts show the Fourier series and its formulas for the coefficients in terms of t rather than x. Personally, I find it easier to change the variable as needed externally and keep the formulas in their simpler and more pristine state.

IMPLEMENTATION

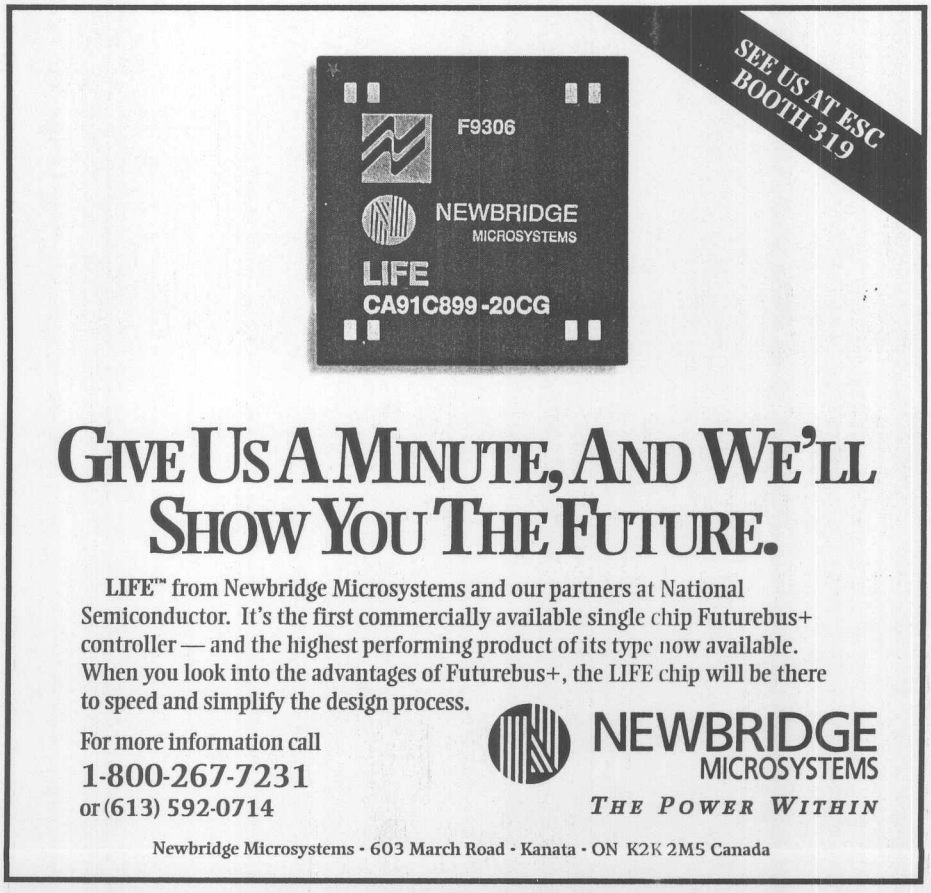
So much for theory. Now, let's put the theory to work with some software. We'll need software of two kinds: one to compute the Fourier series, given x and the coefficients, and the other to compute the coefficients for a given function. The first is by far the easier. We'll assume that the coefficients are given to us as vectors a and b (for once, the C convention of indexing from zero works to our advantage).

FIGURE 4
Sawtooth wave.



A direct translation of Equation 16 gives the code shown in Listing 1. Sure enough, this code gives the desired result and in only four lines of executable code.

Unfortunately, this is also the worst



CIRCLE # 51 ON READER SERVICE CARD

All About Fourier Analysis

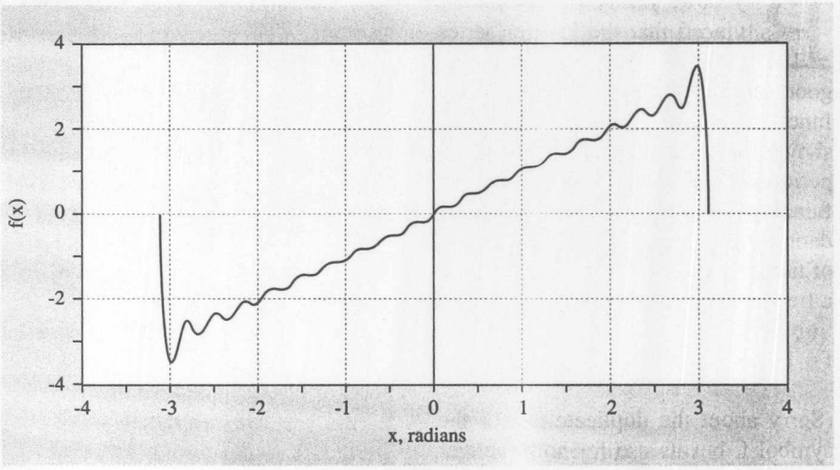
LISTING 1
Fourier first cut.

```
double fourier(double x, Vector &a,
Vector &b, int n){
  double result = a[0]/2.0;
  for(int i=1; i<=n; i++)
    result += a[i]*cos(i*x) +
      b[i]*sin(i*x);
  return result;
}
```

way to compute the series. The problem is that we end up calling the sine and cosine library functions 19 times each. But, you ask, how can I avoid that? The series depends on them. Ah, but they're not independent. Each of the angles is an integral multiple of the original angle, x. We can use this fact and some trig identities to reduce the number of function calls to two.

A pair of trig identities gives us the following:

FIGURE 5
19th order.



$$\sin(a+b)=\sin a\cos b+\cos a\sin b \quad (34)$$
$$\cos(a+b)=\cos a\cos b-\sin a\sin b$$

Let a = x, and b = nx. We can then set up a recursive method to compute the next pair of functions:

$$\sin((n+1)x)=\sin x\cos nx+\cos x\sin nx$$
$$\cos((n+1)x)=\cos x\cos nx-\sin x\sin nx \quad (35)$$

Since we only visit the functions for nx one time each, we can update sin nx

Protected Mode Stress?



Ease into Protected Mode with pmEasy

pmEasy is a complete protected mode environment for embedded systems. It initiates protected mode and provides an application loader, trap handler, error handler, memory manager, debugger support, screen writes and more. *pmEasy* is integrated with low-cost 16- and 32-bit development tools from Microsoft, Borland, and others.

Why struggle developing your own pm environment? *pmEasy* is proven code which lets you focus on your application.

*pmEasy*16 or 32 **$495** including source code

BUY AND TRY 30 day money-back guarantee

MICRO DIGITAL INC *Developer of* smx **1-800-366-2491**
Cypress, CA, USA FAX 714-891-2363 **VISA, MC, AMEX**

CIRCLE # 52 ON READER SERVICE CARD

Real-Time Multitasking

for Microsoft C, Borland C, Borland Pascal

Develop Real-Time Multitasking Applications with *RTKernel!*

RTKernel is a professional, high-performance real-time multitasking kernel. It runs under MS-DOS or in ROM and supports Microsoft C, Borland C++, Borland/Turbo Pascal, and Stony Brook Pascal+. RTKernel is a library you can link to your application. It lets you run several functions/procedures as parallel tasks and offers the following advanced features:

- pre-emptive, event/interrupt-driven scheduling
- number of tasks only limited by RAM
- task-switch time of 6 µ s (33-MHz 486)
- up to 64 priorities (changeable at run-time)
- time-slicing can be activated
- timer interrupt rate of 0.1 to 55 ms
- high-resolution interval timer (1 µ s)
- activate or suspend tasks out of interrupt handlers
- programmable interrupt priorities
- inter-task communications using semaphores, mailboxes, and message passing
- keyboard, hard disk, floppy disk idle times usable by other tasks
- interrupt handlers for keyboard and COM ports included with source code
- supports up to 36 COM ports (DigiBoard and Hostess boards)
- protocols XOn/XOff, DTR/DSR, RTS/CTS
- full support of NS16550 UART chip
- supports math coprocessor and emulator
- fast, inter-network communication using Novell's IPX services
- runs under DOS, LANs, or without OS
- call DOS without re-entrance problems
- resident multi-tasking applications (TSRs)
- runs Windows or DOS Extenders as a task
- supports CodeView and Turbo Debugger
- Kernel Tracer for easy debugging
- ROMable
- **full source code available**
- **no run-time royalties**
- **free technical support by phone or fax**

RTKernel-C 4.0 **$495** **C Source Code:** add **$445**
RTKernel-Pascal 4.0 **$445** **Pascal Source Code:** add **$375**
International orders: add $30 shipping and handling. MasterCard, Visa, check, bank transfer accepted.

FREE DEMO DISK Please request Info Kit E

On Time MARKETING
Professional Programming Tools

In North America, contact:
On Time Marketing
88 Christian Ave. • Setauket, NY 11733
USA • CompuServe 73313,3177
Phone (516) 689-6654 • Fax 689-1172

Outside North America, contact:
On Time Marketing
Karolinenstrasse 32 • 20357 Hamburg
GERMANY • CompuServe 100140,633
Phone +49 - 40 - 43 74 72 • Fax 43 51 96

CIRCLE # 53 ON READER SERVICE CARD

and cos nx in place, like this:

```
temp = sin_x * cos_nx + cos_x * sin_nx;
cos_nx = cos_x * cos_nx - sin_x * sin_nx;
sin_nx = temp;
```

We should start with nx = 0, which gives sin_nx = 0, cos_nx = 1. Listing 2 gives the improved code. It's not four lines long anymore, but it's still quite compact. More to the point, it's about as efficient as you can get. I suppose I should point out that we never use the value of b[0]. But it is easier and less confusing to carry that coefficient around, keeping the indexing as is and wasting one storage location, rather than trying to save that single word and obfuscating the code unnecessarily.

COMPUTING THE COEFFICIENTS

In our two examples, the square wave and sawtooth, we were able to get analytical expressions for the coefficients a_n and b_n. That's because I chose functions such that the integrals in Equations 17 and 18 were integrable in closed form. Clearly, we won't be able to do this in general, and a computer implementation must deal with the issue of how to compute the coefficients. This is where the rubber meets the road because numerical integrations are slow, and we must integrate over the entire range $-\pi..\pi$, once for every coefficient.

The first decision we must make is what kind of integration algorithm we should use. In my previous articles on integration algorithms, I covered all kinds of fancy schemes, including high-order difference methods like Adams-Moulton and single-step methods like fourth-order Runge-Kutta. From the conclusions in those articles, we're tempted to say that the higher the order of integration we use, the better, because we can use a larger step size.

LISTING 2
Fourier improved.

```
double fourier(double x, Vector &a,
    Vector &b, int n){
  double sin_x = sin(x);
  double cos_x = cos(x);
  double sin_ix = 0.0;
  double cos_ix = 1.0;
  double result = a[0]/2.0;
  double temp;
  for(int i=1; i<=n; i++){
    temp =    sin_x * cos_ix + cos_x
      * sin_ix;
    cos_ix = cos_x * cos_ix - sin_x *
      sin_ix;
    sin_ix = temp;
    result += a[i]*cos_ix +
      b[i]*sin_ix;
  }
  return result;
}
```

We'd be wrong, and therein lies a tale.

Many years ago, I was trying to inte-



CIRCLE # 55 ON READER SERVICE CARD



CIRCLE # 58 ON READER SERVICE CARD

grate a complicated function of an angle, Θ, over the entire range, $0..2\pi$. I was looking for an analytical solution, but being unable to find one, I resorted to numerical integration. For simplicity, I chose a trapezoidal integration (second order) with lots of steps, I think about 1,000. I got an answer, but wasn't sure how accurate it was, so I performed the usual sanity check: increase the step size (reduce the number of steps) and try again. I got the same answer. Not just nearly the same; exactly the same. "Hm," says I, "I guess I didn't need as much accuracy as I thought." So I doubled the step size again, and got the same result.

After I got down to only 40 or 50 steps, I was beginning to worry. The result should have started showing the effects of roundoff error long ago, but it was still hanging in there, rock solid. I felt that something must be wrong with my algorithm; it was going to give the same result regardless of the step size. Eventually the roundoff error did start to appear, but only after the number of steps was only 10 or so. This result seemed to defy intuition to such an extent that it was quite awhile before I was able to accept the answer as valid. Finally, however, I figured out what was going on.

If we're going to integrate a function analytically, we must first have a table of its values or at least be able to generate such a table. The integration proceeds in discrete steps with some step size, h. All numerical integration algorithms are based on the notion of fitting a polynomial of some degree through these tabular points. Integrating this polynomial for one step gives us an integral that's a linear combination of adjacent tabular values:

$$\Delta I = h(c_0 f_0 + c_1 f_{-1} + c_2 f_{-2} + \cdots c_n f_{-n}) \quad (36)$$

where f_{-n} stands for the tabular value of f(x) n steps behind the current one. The values of the coefficients c_n depend, of course, on the order of the integration scheme. But we can be sure of one thing: They must add up to unity, because if the function is constant, the integral must be equivalent to hf_0.

Now let's apply Equation 36 over the entire range of the function. We'll end up summing all the individual discreet integrals to get:

$$I = h\sum(c_0 f_0 + c_1 f_{-1} + c_2 f_{-2} + \cdots c_n f_{-n})$$

or:

$$I = h\left(c_0\sum f_i + c_1\sum f_{i-1} + c_2\sum f_{i-2} + \cdots + c_n\sum f_{i-n}\right) \quad (37)$$

CALL FOR PAPERS & SPEAKERS

Embedded Systems Conference

San Jose Convention Center, San Jose, CA

September 12-15, 1995

The Embedded Systems Conference, the world's largest event dedicated to microprocessor and microcontroller-based development, including hardware-related topics, is currently soliciting abstracts for 1-hour lectures, 2-hour workshops, or full-day tutorials. Abstracts for consideration should be submitted by February 6, 1995. All selected speakers are required to write a 6-10 page class text due by June 19, 1995.

For further information, contact Christina Tunnah by phone at (415) 905-2363 or by fax (415) 905-2220

In a general integration of a function that goes on forever, the sums in Equation 37 will all be different because they cover different ranges of the tabular points. But if we're dealing with a periodic function, the values of f_i start repeating each time we cross the limiting value (2π, in our case). All the sums are the same, representing a simple sum of all entries in the table. We can let:

$$S = \sum_i f_i \tag{38}$$

where we understand that this is to include all the tabular points once and only once. Then Equation 37 becomes, simply:

$$I = h(c_0 S + c_1 S + c_2 S + \cdots + c_n S)$$

$$= hS(c_0 + c_1 + c_2 + \cdots + c_n)$$

But we've already said that the coefficients must add up to unity, so we finally get:

$$I = hS = h\sum_i f_i \tag{39}$$

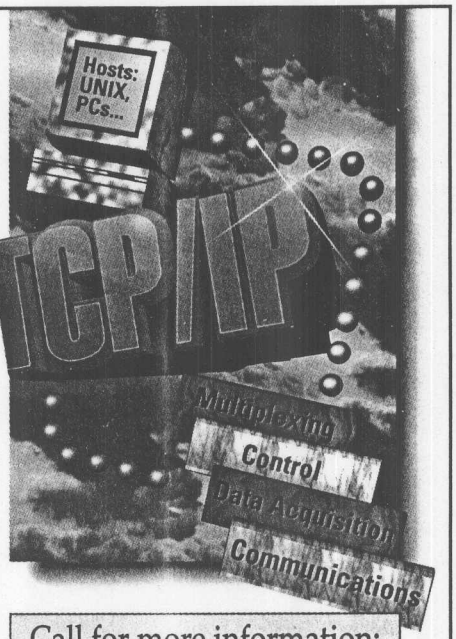
This is a vitally important result. The integration method is seemingly first order, corresponding to a simple sum of all the tabular points. But what we've seen is that it's equivalent to fitting a polynomial of second order, or third order, or, in fact, any order up through n, the number of points, through the tabular data. In other words, the integration method may as well be nth order, which would give us an incredibly accurate algorithm. That's why my simple-minded approach worked so well.

We can use this same approach to get the integrals we need for our Fourier coefficients. No fancy integration schemes are needed; in fact, a fancy scheme would be wasted. One word of caution: If you're tabulating a function of an angle that runs over a range something like $0..2\pi$, each tabular point must be included once and only once. It's tempting to include both the value at zero and the value at 2π to make the function look pretty in a doc-

USNET™ connects your application "with a network", using a real-time embedded processor-independent implementation of TCP/IP!

If your application requires network capability such as Ethernet, serial, or multiple networks --- we have great news for you!

U S Software announces USNET™, a new tool, designed from the ground up, to give you a processor independent implementation of TCP/IP. If your application requires multitasking, USNET has been designed to easily interface with your multitasking system too.



Call for more information:
503-641-8446 • FAX 503-644-2413
Call 800-356-7097

14215 NW Science Park Drive
Portland, OR 97229
U S SOFTWARE.

CIRCLE # 59 ON READER SERVICE CARD

AVOCET SYSTEMS, INC.

C and Pascal Compilers
Simulators
Assemblers
In-Circuit Emulators
EPROM Programmers

Intel 8051 8048 8085 8096
Hitachi 64180 H8
Western 65816
Rockwell 6500 6502
RCA 1802

Motorola 680x0 68300 68HC11 6800 6801 6802 6803 6804 6805 6809
Zilog Z8 Z80 Z180
TI 32010 32020

The Source For Quality Embedded System Tools

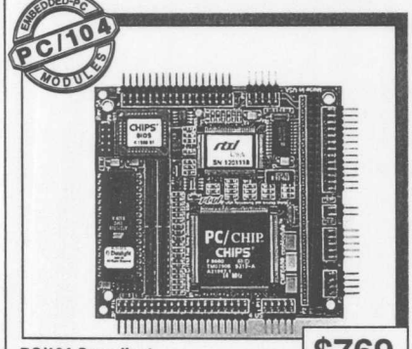
120 Union Street, P.O. Box 490, Rockport, ME 04856
In Maine or outside U.S., call (207) 236-9055
TLX: 467210 Avocet CL • FAX: (207) 236-6713

Call today:
1-800-448-8500

CIRCLE # 60 ON READER SERVICE CARD

Replace Four Conventional PC/104 Modules with *One* CMF8680 cpuModule™

Embedded PC/XT Controller with Intelligent Power Management



PC/104 Compliant
Actual Size: 3.6 x 3.8 x 0.6"

$769

- PC/XT compatibility with 286 emulation
- 14 MHz, 16-bit C&T F8680 CPU
- +5V only; 1.6W at 14.3 MHz, 1W at 7.2 MHz
- Intelligent sleep modes, 0.1W in Suspend
- ROM-DOS and RTD enhanced BIOS
- Compatible with MS-DOS & real-time operating systems
- 1M bootable solid state disk & free software
- 4K-bit configuration EEPROM (2K for user)
- 2M on-board DRAM
- IDE & floppy interfaces
- CGA CRT/LCD controller
- Two RS-232 ports, one RS-485 port
- Parallel, XT keyboard & speaker ports
- Optional X-Y keypad scanning/PCMCIA interface
- Watchdog timer & real-time clock

Expand This Or Any PC/104 System with the

CM106 Super VGA Controller utilityModule™

- Mono/color STN & TFT flat panel support
- Simultaneous CRT & LCD operation
- Resolution to 1024 x 768 pixels
- Displays up to 256 colors

$395

Speed Product Development with the

SK-CM106-X Starter Kit

Your kit includes the CMF8680 cpuModule, CM106 SVGA controller, CM102 keypad scanning/PCMCIA utilityModule, CMF8680 cable kit & VGA monitor cable for just **$1295.**

Additional PC/104 compliant modules from RTD:

- CM104 1.8" hard drive carrier utilityModule
- 12- & 14-bit analog I/O modules
- 12-bit, 4-20 mA analog output modules
- opto-22 & digital I/O modules

For more information on our PC/104 and ISA bus products, call today.

rtd®

Real Time Devices USA
200 Innovation Blvd. • P.O. Box 906
State College, PA 16804 USA
(814) 234-8087 / Fax: (814) 234-5218

RTD Europa • RTD Scandinavia

Real Time Devices is a founder of the PC/104 Consortium.

CIRCLE # 61 ON READER SERVICE CARD

ument. But to do so would result in that point being added twice. Avoid the temptation.

The simple addition of the data points works fine when a function is continuous, but what do we do about discontinuities? The temptation is to include two points at the same value of x, but that would mess up our counting scheme and integration formulas. The key to the right approach is to remember how the series behaves at a discontinuity: It gives a value equal to the mean of the two extreme values. If we provide the same value at a discontinuity, we'll get the right answer.

Of course, that approach only works if the discontinuity lies on a mesh point of the table. If it doesn't, we can only give the values for the two bracketing points and hope for the best. Incidentally, my statements regarding the high accuracy, even when we have few data points, don't apply for this case. It should be clear that fitting a polynomial through discontinuous points is tricky, at best. When a function is discontinuous, there's no substitute for a lot of data points.

THE SUM OF IT ALL

Let's see what our integrals look like when replaced by summations. Equation 17 gives (with a change of index):

$$a_j = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)\cos jx dx$$

As the coefficients (and the Fourier series) are defined, the index is supposed to run to infinity, but of course we must limit it to some finite number, say m.

Now, suppose that f(x) is tabulated with a total of n points, and let i be the index into the table. The step size or granularity of the table is:

$$h = \frac{2\pi}{n} \qquad (40)$$

The first point, for i = 0, corresponds to the point x = -π. The second, for i = 1, corresponds to x = -π + h, and so on. In general:

$$f_i \Leftrightarrow f(-\pi + ih)$$

Our integral is equivalent to:

$$a_j = \frac{h}{\pi}\sum_i f_i \cos j(-\pi + ih)$$

Substituting for h from Equation 40 gives us the multiplying coefficient in terms of n:

$$a_j = \frac{2}{n}\sum_i f_i \cos j(-\pi + ih) \qquad (41)$$

The formula for b_j is similar:

$$b_j = \frac{2}{n}\sum_i f_i \sin j(-\pi + ih) \qquad (42)$$

A straightforward implementation would sum over all i, as in the following pseudocode:

```
for j = 0 to m-1 loop
  a[j] = b[j] = 0
  for i = 0 to n-1 loop
    x = -pi + i*h
    a[j] += f[i] * cos(j * x)
    b[j] += f[i] * sin(j * x)
  end loop
end loop
a = 2*a/m
b = 2*b/m
```

The problem with this approach is that we must have access to the entire array f[] each time through the inner loop. That may not be a problem if the array has reasonable size, but if it is so large that it must be read from a file, we would have to read that file m times. We can improve the algorithm greatly by exchanging the order of summation:

```
a = b = 0
for i = 0 to n-1 loop
  x = -pi + i*h
  for j = 0 to m-1 loop
    a[j] += f[i] * cos(j * x)
    b[j] += f[i] * sin(j * x)
  end loop
end loop
a = 2*a/m
```

```
b = 2*b/m
```

Now each table entry is visited only once in sequential order.

Of course, we have the same efficiency problem as before. As the algorithm is written, we end up calling the sine and cosine functions mn times each—a prohibitively costly approach. But we can use the same trick as before, our double-angle formulas, to compute each trig function from the previous one and reduce the number of functions calls to two. This time, we can apply the trick twice, once for each loop.

Sharp-eyed readers will recognize this process as strength reduction, a technique for loop optimization that we studied several months ago. In this technique, we replace a multiplication involving a loop index by an equivalent addition. Thus, for example, the code:

```
for i = 0 to n-1 loop
  x = -pi + i*h
end loop
```

becomes:

```
x = -pi
for i = 0 to n-1 loop
  x += h
end loop
```

The only difference is that now we are not only applying strength reduction to the multiplication operator, but to trig functions as well, resulting in a huge reduction in overhead.

IS IT FFT YET?

Practical code for the algorithm is shown in Listing 3. The calling convention is simply:

```
fourfit(f, a, b);
```

where the array f contains the input data, and a and b are the output coefficients.

Both f and the coefficient vectors are declared as class `Vector`. As I've defined this class, each vector carries a length parameter, so no additional data in the way of size parameters need be supplied. The order of the fit is determined by the ranks of vectors a and b, which should be of equal size. If they aren't, the algorithm will use the rank of the shorter vector. Remember that the order of the fit includes the constant term a[0], so the size of the coefficient vectors should be the desired order of the fit, plus one. These vectors must be declared and sized in the calling program.

As you can see, I've implemented `fourfit()` as though the tabular data defining the function is stored in an array. However, since each data point is needed only once, you can easily modify the code so each element of ϕ is read from a data file instead. As a matter of fact, if f is treated as a virtual memory array, and if such an array is defined as a C++ class, no code changes will be required at all, other than changing the type of ϕ.

With this algorithm, we are getting very close to the famous Fast Fourier Transform. But no, we're not quite there, yet. In the first place, remember that we're dealing with the Fourier series here, good only for periodic functions, and not the Fourier transform. The latter is a integral relationship. But since we've already seen how integrals get implemented as summations anyway, we're closer than you might think. Because our strength-reduction technique is quite similar to that used in the FFT, we get very good efficiency. In my tests on a 486-based PC, the algorithm processes a 19th-order, 1,024-point transform in well under a second. I call it the "half-fast Fourier transform (FFT/2)."

In one sense, the algorithm is actually more general than the real FFT. Because of the way the loops are implemented, that algorithm requires that n, the number of data points, be a power of two. The FFT/2 algorithm does not—any number will do, as long as it is greater than the number of coefficients.

LISTING 3
Fourier coefficients

```
void fourfit(Vector &f, Vector &a,
    Vector &b){
  int n = f.rank();
  int m = min(a.rank(), b.rank());
  double fcj, fsj, temp;
  double coeff = 2.0/n;
  double h = coeff*pi;
  double s = sin(h);
  double c = cos(h);
  double ci = -1.0;
  double si = 0.0;
  a = 0.0;
  b = 0.0;
  for(int i=0; i < n; i++){
    a[0] += f[i];
    fcj = f[i] * ci;
    fsj = f[i] * si;
    for(int j=1; j < m; j++){
      a[j] += fcj;
      b[j] += fsj;
      temp = fsj * ci + fcj * si;
      fcj = fcj * ci - fsj * si;
      fsj = temp;
    }
    temp = si * c + ci * s;
    ci = ci * c - si * s;
    si = temp;
  }
  a *= coeff;
  b *= coeff;
}
```

We still have one more step to go, extending our result using the Fourier series (a summation) to the limiting case of an integral. We'll do that next time, and that will give us a full-up FFT implementation. We'll also look at a complex-variable version of all the functions. Stay tuned. ESP

Jack W. Crenshaw is a staff scientist at In Vivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs and holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.

by JACK W. CRENSHAW

All About Fourier Analysis, Part II

Our prodigal mathemagician returns to the realm of Fourier analysis, transforms, and other matters of infrequent interest.

In the first part of this series, we considered the whole concept of approximating a function by a collection of other functions.[1] In other words, given a function f(x) defined over some interval, we'd like to replace it by a series:

$$f(x) = a_0\phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) + \cdots + a_n\phi_n(x) + \cdots \quad (1)$$

You're all familiar with the notion of a power series—a special case where the functions $\phi(x)$ are simple powers of x.

Why would we want to replace a single function with a whole (potentially infinite) set of them? The usual reason is that we know something about the new functions, such as how to integrate and differentiate them, that we didn't know about the old one. This is yet another case of the time-honored principle you've seen before in these pages: If you can't solve a problem as it stands, turn it into one you know how to solve.

To make the transformation work, you must know how to compute the coefficients a_n. We looked at a few ways to do that and discovered that the process is particularly easy if the functions are orthogonal, which means that they obey the relationship:

$$\int_r^s \phi_m(x)\phi_n(x)dx = 0 \quad \text{for all } m,n \text{ such that } m \neq n \quad (2)$$

where r and s represent the range over which the original function was defined.

For convenience, we often scale the functions so the integral is unity when m = n, but that isn't really necessary.

It might seem at first glance that Equation 2 gives us an awfully stringent requirement, and you'd be right to wonder if finding such a set of functions is even possible. It is. In fact, a set of sine and cosine functions is such a set (though far from the only one), if we choose the range and frequencies properly. Specifically:

$$\int_{-\pi}^{\pi} \sin mx \sin nx dx = \begin{cases} 0 & (m \neq n) \\ \pi & (m = n) \end{cases}$$

$$\int_{-\pi}^{\pi} \cos mx \cos nx dx = \begin{cases} 0 & (m \neq n) \\ \pi & (m = n, n \neq 0) \\ 2\pi & (m = n = 0) \end{cases}$$

$$\int_{-\pi}^{\pi} \sin mx \cos nx dx = \begin{cases} 0 & (m \neq n) \\ \pi & (m = n) \end{cases} \quad (3)$$

Notice that when m and n are both zero, it doesn't matter what x is; the cosine is constant (equalling one), so the integral is a little different. The extra two in the result causes us some bother for this case, but we live with it.

The notion of using sine and cosine functions to approximate a function is attributed to Jean-Baptiste Fourier, so naturally the subject is called Fourier analysis. Last time, we saw that the

Nance Paternoster

Fourier series is defined to be:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx) \quad (4)$$

(There's that pesky two.) The coefficients a_n and b_n are defined by:

$$a_n = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \cos nx dx \quad (n = 0,1,2...) \quad (5)$$

and:

$$b_n = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \sin nx dx \quad (n = 1,2,3...) \quad (6)$$

The idea of evaluating a potentially very large number of integrals to get the coefficients does not appeal and suggests very slow computations. But we can speed up the process markedly by recognizing that the integrals can be replaced by simple summations, and I showed you the justification for that. Finally, we learned that we can avoid tons of calls to the sine and cosine functions by using the double-angle formulas:

$$\sin(a+b) = \sin a \cos b + \cos a \sin b$$
$$\cos(a+b) = \cos a \cos b - \sin a \sin b \quad (7)$$

This trick lets us reduce a large number of function calls to a mere two. Drawing on that technique, I gave you the code for two functions that compose what I call, with tongue firmly in cheek, the "half-fast" Fourier transform (HFFT). The code is reproduced in Listing 1. Function `fourier` evaluates the Fourier series (Equation 4) from the coefficients, and `fourfit` computes the coefficients from an input function `f(x)`, given in tabular form.

Sharp-eyed readers will notice slight differences between this code and the equivalent listings from the last installment. First, I've removed the number of data points from the argument list of `fourier`. This number can be inferred from the rank of the vector defining f(x). Second, I've replaced an error message testing the ranks of a and b with a C assertion. I've renamed several variables in `fourfit` to make them more consistent and less Fortran-like. The changes don't modify the algorithms, just the readability. Last but far from least, I corrected a serious error in `fourier`—a sign error that made all the results have the wrong sign.

This month, we'll look at ways to further improve the efficiency of the computation and use it in different ways.

This month, we'll look at ways to further improve the efficiency of the computation and use it in different ways. This will lead us to the discrete Fourier transform (DFT). Then, we'll look at what happens when we extend the concept of a series (a sum of terms) to an integral. This will give us the Fourier transform. Unlike the Fourier series, the Fourier transform allows us to extend the concepts into the realm of nonperiodic functions.

SERIES OR INTEGRAL?

That last statement calls for some clarification. From some of the responses I've gotten to Part 1 of this series, I've realized that many people are confused over the difference between the DFT and the FFT and between the Fourier series and the Fourier integral. Here's the difference in a nutshell: The Fourier series works only for periodic functions. Because all of its terms contain factors like sin nx,

All About Fourier Analysis, Part II

LISTING 1
The "half-fast Fourier transform."

```
// Evaluate Fourier series
double fourier(double x, Vector &a,
    Vector &b){
  int m = a.rank();
  assert(a.rank() == b.rank());
  double temp;
  double cos_x = cos(x);
  double sin_x = sin(x);
  double cos_ix = -1.0;
  double sin_ix = 0.0;
  double result = a[0]/2.0;
  for(int i=1; i<m; i++){
    temp = sin_x * cos_ix + cos_x *
      sin_ix;
    cos_ix = cos_x * cos_ix - sin_x *
      sin_ix;
    sin_ix = temp;
    result += a[i]*cos_ix + b[i] *
      sin_ix;
  }
  return result;
}


// Compute Fourier coefficients
void fourfit(Vector &f, Vector &a,
    Vector &b){
  int n = f.rank();
  int m = a.rank();
  assert(a.rank() == b.rank());
  assert(m <= n/2);
  double fc, fs, temp;
  double coeff = 2.0/n;
  double h = coeff*pi;
  double cos_h = cos(h);
  double sin_h = sin(h);
  double cos_x = -1.0;
  double sin_x = 0.0;
  a = 0.0;
  b = 0.0;
  for(int i=0; i < n; i++){
    a[0] += f[i];
    fc = f[i] * cos_x;
    fs = f[i] * sin_x;
    for(int j=1; j < m; j++){
      a[j] += fc;
      b[j] += fs;
      temp = fs * cos_x + fc * sin_x;
      fc = fc * cos_x - fs * sin_x;
      fs = temp;
    }
    temp = sin_x * cos_h + cos_x *
      sin_h;
    cos_x = cos_x * cos_h - sin_x *
      sin_h;
    sin_x = temp;
  }
  a *= coeff;
  b *= coeff;
}
```

the series is going to repeat itself every 2π radians, no matter what. Your input function f(x) may not be repetitive; it may be only defined on the interval $\pi..\pi$ (or some appropriately scaled equivalent) and be undefined outside that range.

Regardless of how the function f(x) behaves outside this range, the approximation embodied in the Fourier series is going to repeat itself at 2π intervals. To the mathematics, the values $-\pi$ and $+\pi$ are the same point. The values for your function at these endpoints had better be equal as well. You can apply Fourier analysis to functions in which the values at $-\pi$ and $+\pi$ are different, and many people have. But you may not like the results because the mathematics tries to wrap the function around at those endpoints, and it will treat any differences between the two ends as a jump discontinuity. You saw in Part 1 that the Fourier series handles discontinuities well. But you've also learned to expect "ringing" and edge transients such as Gibb's phenomenon in the vicinity of those discontinuities. If you try to use the Fourier series to approximate a segment of a general, nonperiodic function, you will see those effects, and the series will not give you a good approximation to the function, even within the range over which it's defined.

The Fourier integral, on the other hand, is another matter altogether. In general, it will contain components of all frequencies, and they won't all be commensurate. So, there will be no value of x for which the function repeats itself. Fourier transforms apply to functions that are nonperiodic, whereas the Fourier series cannot. That's the fundamental difference between the two. We'll go into all this in detail, but the distinction between the two cases will help clear some of the fog in what follows. For now, we'll continue with the case of the Fourier series, not the integral.

Between the DFT and the FFT, there is no difference in functionality. If both are done correctly, the two algorithms should yield identical results. The difference is internal and, as the name implies, comes from seeking ways to speed up the DFT. Make no mistake, we're not talking about minor improvements in efficiency here. The FFT is not just faster, it's profoundly faster because as the number n of data points is increased, the time required to perform the DFT increases by order(n^2), while that of the FFT increases only by order(n log n). For large n, that difference can be staggering. My cherished "half-fast Fourier transform" is really a DFT, so perhaps even that whimsical name is far too optimistic.

HOW MANY POINTS?

Mathematically, the "input" function f(x) is a continuous function, defined over some range that we've taken here to be $-\pi$ to $+\pi$. Also mathematically, the number of coefficients is infinite. The series never quits except for very special functions. In the real world, of course, things look quite different. We can expect to be given the function not as an analytical one, but as a table of points. (If we had an analytical expression for f(x), we'd use it). What's more, we can expect to get not only a table, but a table based on evenly-spaced divisions in x; the whole notion of using the double-angle formulas depends on this. Finally, we must recognize that we can't carry around an infinite number of coefficients, so we end up truncating the series at some order. This step means that our series no longer precisely represents the input function, but is an approximation to it.

All About Fourier Analysis, Part II

Let's call the number of input points n and the order of the approximation, or the highest frequency we retain, m. (Careful—these are not the same m and n as in Equations 1 through 6. The usage is conventional and the functions in Listing 1 are consistent with it.)

When I first wrote the functions `fourier` and `fourfit`, it was to use the concept of a series approximation precisely as Fourier intended it: to approximate a periodic function f(x), given a single value of x. I was building dynamic simulations at the time and wanted to approximate things like aerodynamic effects. Since the angles associated with aerodynamic effects are, after all, angles and inherently periodic, such effects are natural candidates for this kind of approximation. The idea was to execute function `fourfit` offline, using it only once to give the needed coefficients. These would then be appropriately truncated and stuffed into the simulation as data. During a computation, function `fourier` would be called as needed for specific values of x to give me the approximated forces and torques. Keep these functions in your toolkit because they are very useful for this kind of application, and you won't find function `fourier` in any textbook I've ever seen. Its use to evaluate the series at a single point is rather special.

When using these routines for their intended purpose, I wanted as accurate a representation as possible within performance limits. Since function `fourfit` would typically be run only once or a few times for each new simulation, I really didn't care how long it took to execute. If you'd asked me how many points I wanted in the table for f(x), I would surely have said, "As many as possible."

In the execution of the simulation, on the other hand, time was critical, so I wanted to use as few terms in the series as possible. Typically, I'd find that fifth or seventh order was adequate. If you'd asked how large m, the order, should be, I'd say, "Something more than seven." I needed to see only enough terms to see how the coefficients were decreasing with order, so I'd know where to chop off the series. In my case, n was much, much greater than m.

If we're doing image enhancement or data compression, we don't want to see just one point of the transformed image, but the whole thing.

You must understand, though, that this isn't the way most people use Fourier analysis. In my article, "The Alphabet from S to Z,"[2] we saw that, while transforms such as the Laplace transform and z-transform can be used in both directions, control systems engineers often stop halfway. Except when Laplace transforms are being used to solve differential equations, we find little need to reconstruct the function we started with. Control systems engineers only want to see the transfer function in s or z space because they can infer from the nature of that function what the behavior of their system is going to be. They don't need or want the inverse transform.

It's the same way with Fourier transforms. Communications engineers and others who use such transforms often don't want to reconstruct the function f(x). After all, they already had it once. What they're interested in is the frequency and phase content of the signal. For that purpose, they need to see as much of the spectrum as possible. If you ask those engineers how many coefficients they need to see, they're likely to say, "All I can get." In this case, that term represents a specific number. According to Shannon's theorem, when a given signal is sampled at a certain frequency, no information contained in that signal can be retrieved if it's more than half the sampling frequency. Producing a tabular representation of a continuous function with, say, n points is equivalent to sampling at a frequency of n times the base frequency. Thus m can be no larger than $n/2$. Since we have both sine and cosine functions, we'll get out n coefficients in total. And the folks who are looking at frequency spectra will want to see every one of them.

A similar situation holds in the opposite direction. If we're doing something like image enhancement or data compression, we don't want to see just one point of the transformed image, but the whole thing. These considerations represent the major difference between functions `fourier` and `fourfit` as they were intended to be used and the way most applications use the same formulas. In my case, the forward transform (`fourfit`) was a many-to-few transformation, while the reverse transform (the series evaluation, `fourier`) was a few-to-one. Most modern applications involve a many-to-many transformation. In fact, the number of output data items is almost surely the same as the number of input data items.

The form of `fourfit` already fits this usage; we need only make the vectors the correct length (the coefficient vectors should be half as large as the function vector). We could get the other half (the vector version of `fourier`) simply by calling `fourier` n times, but that would hardly be efficient. But if you look at the form of `fourier`, you'll see that we don't actually use the value of x itself, but rather its trig functions. We can compute these functions in recurrence relations, just as we have in `fourfit`. The end result looks awfully

All About Fourier Analysis, Part II

familiar; it has almost exactly the same form as `fourfit`. The routine is shown in Listing 2. A side-by-side comparison of `inv_four` and `fourfit` shows that they're almost the same. This, of course, is as it should be; we saw in "The Alphabet from S to Z" that the inverse Laplace transform looks almost like the forward one. The routines will look even more alike when we take the next step, which is to convert them to use complex arithmetic.

THE COMPLEX SERIES

As any electrical engineer knows, the relationship between trig functions and complex numbers is very close indeed. Specifically, a pair of identities gives:

$$\cos x = \tfrac{1}{2}(e^{ix} + e^{-ix})$$
$$\sin x = \tfrac{-i}{2}(e^{ix} - e^{-ix}) \qquad (8)$$

Substituting these relationships into Equation 4 gives:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[\frac{a_n}{2}(e^{inx} + e^{-inx}) - \frac{ib_n}{2}(e^{inx} - e^{-inx})\right]$$

Collecting terms gives:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left(\frac{a_n}{2} - \frac{ib_n}{2}\right)e^{inx} + \sum_{n=1}^{\infty}\left(\frac{a_n}{2} + \frac{ib_n}{2}\right)e^{-inx} \qquad (9)$$

We can write this equation as:

$$f(x) = c_0 + \sum_{n=1}^{\infty} c_n e^{inx} + \sum_{n=1}^{\infty} d_n e^{-inx} \qquad (10)$$

where:

$$c_0 = \frac{a_0}{2}$$
$$c_n = \frac{a_n}{2} - \frac{ib_n}{2} \qquad (n > 0) \qquad (11)$$

and:

$$d_n = \frac{a_n}{2} + \frac{ib_n}{2} \qquad (n < 0)$$

In the third sum of Equation 10, let's replace n by -n. Then the sum becomes:

$$\sum_{n=-\infty}^{1} d_n e^{inx}$$

Equation 10 now takes the form:

$$f(x) = +\sum_{n=-\infty}^{-1} d_n e^{inx} + c_0 + \sum_{n=1}^{\infty} c_n e^{inx} \qquad (12)$$

The first sum takes us over all the negative integers and the second one all the positive ones. The constant term takes care of n = 0. What's more, the exponential in each element is identical. This holds for the constant term as well, since $e^{inx} = 1$ when n = 0. We can make the symmetry complete by letting $d_n = c_{-n}$, implying that:

$$c_{-n} = \frac{a_n}{2} + \frac{ib_n}{2} \qquad (13)$$

If a_n and b_n are real, as we've assumed, the c_n for negative n are simply the complex conjugates of their positive counterparts.

Our Fourier series now takes the very simple form:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx} \qquad (14)$$

In this representation, even that pesky power of two for the constant term a_0 has gone away. Equation 14 has a certain elegance about it, don't

AMX™ The Real-Time Multitasking Kernel

NEW DOS Compatible File System TCP/IP

680x0, 683xx
i960® family
R3000, LR330x0
80x86/88 real mode
80386 protected mode
Z80, HD64180

Features

- Full-featured, compact ROMable kernel with fast interrupt response
- Preemptive, priority based task scheduler with optional time slicing
- Mailbox, semaphore, resource, event, list, buffer and memory managers
- InSight™ Debug Tool is available to view system internals and gather task execution statistics
- Configuration Builder utility eases system construction
- Supports inexpensive PC-hosted development tools
- Comprehensive, crystal clear documentation
- No-hidden-charges site license
- Source code included
- Reliability field-proven since 1980

For a free Demo Disk and your copy of our excellent AMX product description, contact us today. Phone: (604) 734-2796 Fax: (604) 734-8114

Count on KADAK.
Setting real-time standards since 1978.

KADAK Products Ltd. 206 - 1847 West Broadway, Vancouver, BC, Canada V6J 1Y5

AMX is a trademark of KADAK Products Ltd. All trademarked names are the property of their respective owners.

CIRCLE # 22 ON READER SERVICE CARD

LISTING 2
Generating a result vector.

```
// Regenerate Fourier function
void inv_four(Vector &a, Vector &b,
    Vector &f){
  double cos_jx, sin_jx;
  double temp, result;
  int n = f.rank();
  int m = a.rank();
  assert(a.rank() == b.rank());
  double h = 2*pi/n;
  double cos_h = cos(h);
  double sin_h = sin(h);
  double cos_x = -1.0;
  double sin_x = 0.0;
  for(int i=0; i<n; i++){
    cos_jx = 1.0;
    sin_jx = 0.0;
    result = a[0]/2.0;
    for(int j=1; j<m; j++){
      temp = sin_x * cos_jx + cos_x *
        sin_jx;
      cos_jx = cos_x * cos_jx - sin_x *
        sin_jx;
      sin_jx = temp;
      result += a[j]*cos_jx + b[j] *
        sin_jx;
    }
    f[i] = result;
    temp = sin_x * cos_h + cos_x *
      sin_h;
    cos_x = cos_x * cos_h - sin_x *
      sin_h;
    sin_x = temp;
  }
}
```

you think? We haven't really changed the nature of the series or even saved any storage or arithmetic requirements, but the series certainly looks simpler, and the math is easier to work with, assuming your programming language supports complex arithmetic. C does not, but most C++ vendors are supplying a complex number class as part of their libraries, and ANSI is working to establish a standard class. See P.J. Plauger's "Complex Made Simple" for the latest information.[3]

At first glance, it may seem that we've required even more storage because the index of c_n must run over all integers, where that of a_n and b_n ran only over the nonnegative ones. However, the c_n are not all independent: The values for positive and negative n are complex conjugates of each other:

$$c_{-n} = c_n^* \tag{15}$$

So we need to evaluate and store coefficients only for nonnegative values.

Let's derive expressions for the coefficients c_n. We can get these expressions directly from the definition of the series, as we originally found a_n and b_n, but you might be more convinced we're on the right track if you see them derived from Equations 11. For c_0, we have:

$$c_0 = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)dx \tag{16}$$

Equation 5 gives the definitions for a_n. Substituting the identity from Equation 8 gives:

$$a_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)(e^{inx} + e^{-inx})dx$$

Similarly, for b_n:

$$b_n = \frac{-i}{2\pi}\int_{-\pi}^{+\pi} f(x)(e^{inx} - e^{-inx})dx$$

Now, combining these in Equation 11, we get:

$$c_n = \frac{1}{2\pi}\left[\int_{-\pi}^{+\pi} f(x)(e^{inx} + e^{-inx})dx - i(-i)\int_{-\pi}^{+\pi} f(x)(e^{inx} - e^{-inx})dx\right]$$

(Watch out for all those -i terms. Their product is minus one.) Combining terms gives:

BSO/TASKING
8051
8051
The Best 8051 Software Tools on the Market.
All 8051 manufacturers supported
Highly optimized C and PL/M compilers with assembler, linker and locator
CrossView® Windows debugger with Target ROM or Simulator
Complete ANSI C and runtime libraries including efficient floating point support
Available for DOS and Unix
Small high performance RTOS
Context sensitive help and on-line user manuals
EDE gives you access to all tools through a common interface
Toll-free hot line support
BOSTON SYSTEMS OFFICE TASKING
Discover how fast your application can be.
Ask for your free demo disk and our competitive benchmarks
1-800-458-8276
Fax 617-320-9212
TASKING Europe +31 33 558584
TASKING Japan +81 334 050511
All trademarked names are the property of the respective owners
CIRCLE # 23 ON READER SERVICE CARD

All About Fourier Analysis, Part II

$$c_n = \frac{1}{4\pi}\left[\int_{-\pi}^{+\pi} f(x)(e^{inx} + e^{-inx} - e^{inx} + e^{-inx})dx\right] = \frac{1}{4\pi}\left[\int_{-\pi}^{+\pi} 2f(x)e^{-inx}dx\right]$$

or:

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)e^{-inx}dx \qquad (n > 0) \tag{17}$$

Similarly, the case for n < 0 gives:

$$c_{-n} = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)e^{+inx}dx$$

which becomes the same as Equation 18 if we replace -n by n. Since this formula also matches the special case n = 0, we end up with one expression for all n.

We now have a rigorous definition of the Fourier series in its complex form. It is:

$$f(x) = \sum_{n=-\infty}^{+\infty} c_n e^{inx} \tag{18}$$

where:

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)e^{-inx}dx \tag{19}$$

Armed with suitable classes for complex numbers and their vectors, we can rewrite the code of Listings 1 and 2 to use these equations. The results are shown in Listing 3. Small, aren't they? Function `fourier` now contains eight executable lines of code. That's a lot of processing in a small package.

IS IT DFT YET?

The listings deserve a bit of explanation. First, I've clearly defined a class `C_Vector` that is virtually identical to class `Vector`, except its elements are complex. I'm using the Borland class for complex numbers. In `fourier`, I generate the function value on the assumption that it's real, which was the same assumption Fourier made. Because of the symmetry in c_n for this case, we don't need to store the elements for negative n, which saves us half of the storage. Finally, take a look at the listings for `fourfit` and `inv_four` side by side. You'll see that they're virtually identical. So, do we finally have a discrete Fourier transform? Yes and no. In one sense, we have more than the DFT since these functions are more flexible. Remember, they were originally written to support the concepts of the Fourier series as Fourier himself defined it. That is, we express a periodic function f(x) in terms of its sinusoidal components, with the purpose of approximating the original function or otherwise doing tricks with it.

Remember, function `fourfit` was designed to be a many-to-few transformation; I expected to generate a set of coefficients much smaller than the number of points in the original function. Function `fourier` was a few-to-one conversion, but there's nothing except Shannon's limit to keep us from using the routines with larger vectors. That limit requires that the order of c be no larger than half the number of samples. Since c is complex, the number of elements in c is precisely the same as those in the original function vector. A typical DFT algorithm is written based on the assumption that this will always be the intended use. If this is what you want, and you're only interested in dealing with real functions, you can simply declare vectors f and c with dimensions n and $n/2$, respectively, and you've got your DFT.

Real functions, did you say? Are there any other kind? Of course there are. Although we've been working with f(x) as though it were always real, there's really nothing in the math that depends on that assumption. Equations 18 and 19 work just as well if f(x) is complex. The only problem is that the elements of c for positive and negative

LISTING 3
HFFT, complex version.

```
// Evaluate Fourier series
double fourier(double x, C_Vector & c){
  int m = c.rank();
  complex w = exp(complex(0, x));
  complex wi = complex(1, 0);
  double result = real(c[0]);
  for(int i=1; i<m; i++){
    wi *= w;
    result += 2.0 * real(c[i] * wi);
  }
  return result;
}

// Compute Fourier coefficients
void fourfit(Vector &f, C_Vector &c){
  int n = f.rank();
  int m = c.rank();
  assert(m <= n/2);
  complex w = exp(complex(0, 2.0 *
    pi/n));
  complex wi = complex(1, 0);
  complex wij;
  c = 0;
  for(int i=0; i < n; i++){
    wij = complex(1, 0);
    for(int j=0; j < m; j++){
      c[j] += f[i] * wij;
      wij *= wi;
    }
    wi *= w;
  }
  c /= n;
}

// Regenerate Fourier function
void inv_four(C_Vector &c, Vector &f){
  int n = f.rank();
  int m = c.rank();
  complex w = exp(complex(0, 2.0 *
    pi/n));
  complex wi = complex(-1, 0);
  complex wij;
    f = 0;
  for(int i=0; i<n; i++){
    wij = complex(1, 0);
    for(int j=0; j<m; j++){
      f[i] += 2.0 * real(c[j] * wij);
      wij *= wi;
    }
    wi *= w;
  }
}
```

If we can't yet call the functions I've shown DFTs, it's because we haven't yet defined the Fourier transform itself.

indices will no longer be complex conjugates, so we must store the values for the full range. This, in turn, gets us into the issue of how to organize the vector; C++ doesn't allow negative indices. Programming issues aside, but the approach is practical. Although we'll now have twice as many elements to store, we'll still have an equal number in both vectors. Many DFTs are written to work with complex functions. The only problem with that approach is that 99% of the time, you'll be working with functions that are real, so you'll be wasting half the data storage (and using four times the computing time) for no good reason. This causes a certain bother.

I've seen programs that solve the dilemma by transforming two real functions together, putting them in the real and imaginary parts of the complex vector f. But that approach strikes me as something of a kludge—a way to do something with the storage that's sitting there, allocated. How often can I expect to be dealing with two totally different functions, but two I happen to have tabulated with the same number of data points? Not very often, I'll wager. For most cases, I prefer the routines of Listing 3, which assume that f(x) is real. However, the symmetry between `fourfit` and `inv_four` is more nearly perfect if we use complex values for both vectors.

HOME ON THE RANGE

If there's a reason we can't yet call the functions I've shown DFTs, it's because we haven't yet defined the Fourier transform itself. It's about time we did.

To do so, we must first talk about the range over which the function f(x) is defined. As we've been using the Fourier series, we've taken that range to be $-\pi \ldots +\pi$. Such a range is reasonable when we're talking about an angle that's naturally repetitive over that interval. Even then, we might prefer a range beginning at rather than centered on zero. Again, in the case of an angle, it really makes little difference which range we choose, since the function repeats itself on an interval of 2π in any case. As a start, suppose that the function given to us is not centered at the origin, but begins there. We can easily translate the range by a substitution of variables. Let:

$$y = x + \pi \qquad (20)$$

so, equivalently:

$$x = y - \pi \qquad (21)$$

Let's define a new function, g(y), such that:

$$g(y) = f(y - \pi) \qquad (22)$$

Our Fourier series becomes:

$$g(y) = f(y-\pi) = \sum_{n=-\infty}^{+\infty} c_n e^{in(y-\pi)}$$

$$= \sum_{n=-\infty}^{+\infty} c_n e^{inx} e^{-in\pi}$$

The last exponential is independent of y, so we can think of it as merely multiplying c_n. Define a new coefficient:

$$c_n' = c_n e^{-in\pi} \qquad (23)$$

Then our series looks exactly like it did before, except for a prime over the coefficient, as follows:

BSO/TASKING
68HC08



Migrate to Performance with the Real HC08 Solution.

Optimized ANSI C compiler supports three memory models

Powerful Motorola compatible macro assembler, linker and locator

CrossView® Windows debugger with Simulator or Emulator environment

Available for DOS and Unix

Complete ANSI C and run-time support libraries

Context sensitive help and on-line user manuals

EDE gives you access to all tools through a common interface

Toll-free hot line support

BOSTON SYSTEMS OFFICE TASKING

Put us to the test.

Ask for your free demo disk and our competitive benchmarks

1-800-458-8276

Fax 617-320-9212

TASKING Europe +31 33 558584

TASKING Japan +81 334 050511

All trademarked names are the property of the respective owners

CIRCLE # 24 ON READER SERVICE CARD

$$g(y) = \sum_{n=-\infty}^{+\infty} c'_n e^{iny} \qquad (24)$$

That prime is a hint that the coefficient array will be different. The difference is embodied in Equation 23. From the definition of c_n, we can write:

$$c'_n = \left[\frac{1}{2\pi} \int_{-\pi}^{+\pi} f(x) e^{-inx} dx \right] e^{-in\pi}$$

or:

$$c'_n = \frac{1}{2\pi} \int_{-\pi}^{+\pi} f(x) e^{-in(x+\pi)} dx$$

Making the same substitution for y and noting the values of y when x is at its limits, we get:

$$c'_n = \frac{1}{2\pi} \int_{0}^{2\pi} g(y) e^{-iny} dx \qquad (25)$$

This equation also has precisely the same form as Equation 19, except for the range of integration. The implication is this: The algorithm doesn't really care what you define the range to be (even though a range is implicit in the formulation of the computer programs). After all, a periodic function is a periodic function, meaning that it will go on forever in both directions, repeating at intervals of 2π. The point you pick to be the starting point is rather arbitrary and depends more on your convenience than any deep mathematical requirement. Just be consistent, and everything will be fine.

On the off chance that you do need to change the coefficients after they were generated, Equation 23 gives what you need. In general, an offset through any angle is equivalent to a rotation in the complex plane. But a rotation through an angle like π is very special; it really isn't so much a rotation as a simple sign change. This is most obvious if we look at the real rather than complex series. We know that:

$$\sin(x + n\pi) = \sin x \cos n\pi + \cos x \sin n\pi$$
$$= \sin x \cos n\pi$$

Similarly:

$$\cos(x + n\pi) = \cos x \cos n\pi$$

Finally, cos $n\pi$ obeys the simple rules:

cos $n\pi$ = 1 (n even)
cos $n\pi$ = -1 (n odd)

To get one set of coefficients from the other, change the sign of every odd coefficient. We can easily see that this scheme works for the special case of a square wave. If we define the function as odd (by putting the transition at the origin), we get no cosine terms, and the sine terms are zero except for odd orders. Thus, every nonzero coefficient of the square wave gets negated when we shift the origin by π. And since every term is negated, the resulting value of the function as evaluated using the series is negated as well. That's precisely the result we should get, since shifting a square wave by π is indistinguishable from simply negating it.

LAW OF AVERAGES

Throughout this discussion, one point was probably painfully obvious to the mathematicians among us, but perhaps not so for the rest. Going back to the real series once more, consider the definition of a_0, or rather, $a_0/2$. From Equation 5:

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) dx$$

so:

$$\frac{a_0}{2} = \frac{1}{2\pi} \int_{-\pi}^{+\pi} f(x) dx \qquad (26)$$

Here we're integrating over a range of

BSO/TASKING
80C196
intel
The New Standard in C196 Tools.
Highly optimized C compiler with Intel compatible assembler, linker and locator
ANSI C and run-time libraries supplied including floating point support
ChipView® debugger with Target ROM environment
Small high performance RTOS
On-line user manuals with context sensitive help
EDE gives you access to all tools through a common interface
Support hot line only a toll-free call away
BOSTON SYSTEMS OFFICE TASKING
Put the standard to work for you.
Ask for your free demo disk and our competitive benchmarks
1-800-458-8276
Fax 617-320-9212
TASKING Europe +31 33 558584
TASKING Japan +81 334 050511
All trademarked names are the property of the respective owners
CIRCLE # 25 ON READER SERVICE CARD

All About Fourier Analysis, Part II

2π, then dividing by that same factor. Mathematicians will recognize immediately that this integral is the simple average of f(x). So the constant term in the Fourier series is simply the average of the function. Similarly, every other coefficient is also an average. The formula for the complex coefficient:

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)e^{-inx}dx$$

also represents an average, this time of the function:

$$f(x)e^{-inx}$$

This inner nature of the integrals will be important when we deal with intervals other than 2π, which we'll do next.

FUNCTIONS OF TIME

The range $-\pi..+\pi$ may be all right for angles that naturally repeat on that interval. But a goodly portion of our work ends up dealing with functions of time, and few are going to be so accommodating as to repeat every 6.28 seconds. To deal with functions of time (or any other measurements than angles), we need a much more flexible definition of the Fourier series and coefficients.

Fortunately, we can easily accommodate the changes based on what we've already learned. Consider a function of time with period T. Let:

$$x = \omega t \tag{27}$$

and we'll find out what value w must have. For things to work properly, we require that when $t = T/2$, x will be π (and when t = T, x will be 2π). Solving for ω gives us:

$$\omega = \frac{2\pi}{T} \tag{28}$$

The ratio $1/T$ is called the frequency f, and many formulations of the Fourier transform use f instead of ω. I'll use f sparingly here, partially to avoid confusion between the frequency f and the function f(x). Substituting our new definition for x into Equations 18 and 19 gives:

$$f(t) = \sum_{n=-\infty}^{+\infty} c_n e^{in\omega t} \tag{29}$$

where:

$$c_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t)e^{-in\omega t}dt \tag{30}$$

Looking at Equation 29, we see that each term is a sinusoid with a frequency of n times the fundamental frequency. Each coefficient c_n defines the amplitude and phase of the harmonic associated with its corresponding frequency nf. In other words, c_n gives us the spectrum of f(t).

With that thought, we begin to see the concept and value of Fourier analysis. This is no longer about approximating a function. Fourier analysis is about analyzing a signal to determine its frequency content. That can be important in a wide variety of fields, from communications to stereo to image processing to data compression.

For these kinds of applications, the set of coefficients c_n become far more than just the values we use to regenerate the signal. In modern analysis, we often don't bother to regenerate it since we already have the original. For spectral analysis, the coefficients are the message, the thing we're interested in seeing.

The change from using the Fourier series to doing Fourier analysis is both trivial and profound. It involves a shift, not so much in how we do things, but in how we think of the results. When we're using the equations to approximate a function using the Fourier series, the vector c is simply a set of coefficients whose values we need to get the right fit. When we're doing Fourier analysis, c is the desired output or spectrum that describes the function.

Look at it this way—we began with a continuous function of time, f(t). We expressed that function as a table of values or, more accurately, a vector. Strictly speaking, once we've converted the function to tabular form, we can no longer think of it as continuous, but we do. On the other hand, our algorithm produces the output vector c, which our brains still want to think of as being a discrete set of coefficients, mere byproducts of the algorithm. But just as f(t) can be a continuous function, a table of values, or a vector, we can think of c as being a function in its own right—a function of ω rather than time. Just as we've seen the Laplace transform convert a function from the time domain to the s-domain or the z-transform to the z-domain, we now have another domain to play with: the frequency domain of Fourier analysis.

Over the years, people have come to adopt the universal symbol h (or H) to represent transforms. We can think of the transformed functions not so much as new functions, but as different representations of the same function. Conceptually:

h(t) <=> H(w) <=> H(s) <=> H(z)

Which representation we get depends on which transform we applied to get it. If nothing else, this notation, in which the nature of the function is implied by its argument, saves time trying to think up new symbols for each representation.

OK, so we can think of c as being a function of ω. Does that mean we can also think of it as being continuous, like f(t)? Well, no, we can't, not when it's derived from a periodic function. We have no need to do so, since the only frequencies that appear in the function, appear at intervals of the fundamental, which we might call ω_0. In the jargon of spectral analysis, there's no energy at other frequencies, so the function representing the spectrum really is discontinuous; energy appears as infinitesimally wide spikes at the harmonic frequencies, and the amplitude is zero, otherwise. Figure 1 shows

a plot of the amplitude vs. frequency for the square wave we've used as a test case. In Figure 1, I've normalized the frequency to the fundamental ω_0.

Still, this whole new way of looking at things certainly raises our curiosity: Do functions exist for which $H(\omega)$ is continuous? The answer is, yes indeed, and that brings us to the notion of the Fourier integral. Not surprisingly, if periodic functions lead only to functions in the frequency domain that are discontinuous, we must look to the realm of nonperiodic functions for anything else. It's interesting, though, that functions exist that have only discrete frequencies, but are still aperiodic. Hamming gives a very simple example:[4]

$$f(t) = \sin \omega t + \sin \sqrt{\omega}\, t \qquad (31)$$

As you can see, this function contains only two pure sine waves. In a plot of the frequency spectrum, it would have only two lines where the values were anything but zero. But the two frequencies are not commensurate. Therefore, once started up with a given set of initial conditions, the function will never, ever, repeat itself. Each time one of the sine waves passes through a certain phase angle, say zero, the other wave will have a different phase than it has ever had before. This function is not periodic.

We can never represent such a function as a Fourier series because no combination of harmonics can ever represent both waves. If we try, we'll end up with a discontinuity at the edges of any interval we try to define. And as you know, discontinuities lead to ringing, Gibb's phenomenon, and so on. A Fourier analysis, based on any finite interval, will yield frequency components that don't really exist in the original function.

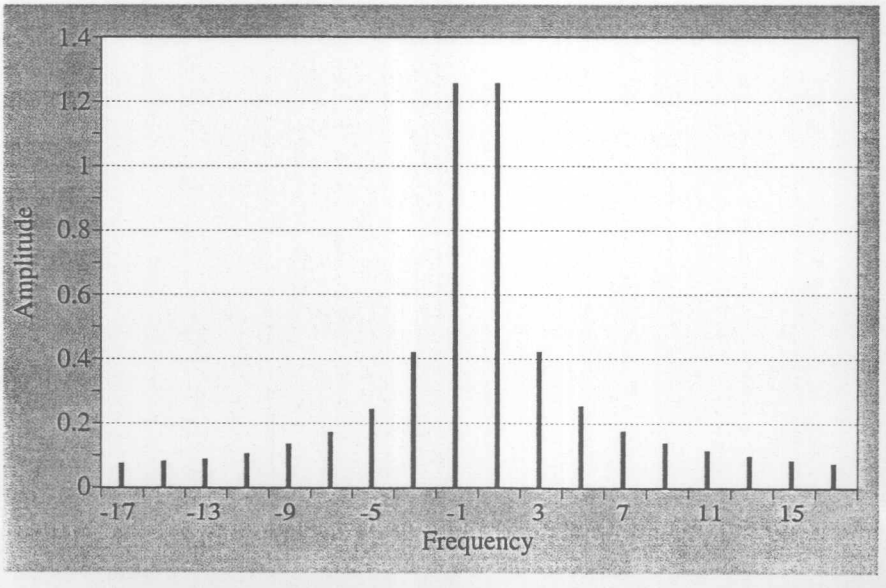
If we're to have any hope of representing a function like this one, we must turn to a representation that includes all frequencies, not just those that are harmonics of a fundamental. And that can only be done with an integral, not a summation.

THE FOURIER INTEGRAL

The question is, how do we get from a series to an integral representation? Does such a thing exist, or are we just dreaming? I can give you four approaches. Each of them has some merit, but none of them completely satisfies. Together, they make a powerful argument.

The first argument recognizes that

FIGURE 1
Spectrum for a square wave.



BSO/TASKING
80C166
SIEMENS
Harness the Power of the 166.
Optimized C compiler supports 165/166/167 family, includes assembler, linker, locator and all libraries
CrossView® Windows debugger with Target ROM environment
Available for DOS and Unix
Small high performance RTOS
Evaluation boards for all family members
On-line user manuals with context sensitive help
EDE gives you access to all tools through a common interface
Support hot line only a toll-free call away
BOSTON SYSTEMS OFFICE
TASKING
The hottest tools for this fast microcontroller.
Ask for your free demo disk and our competitive benchmarks
1-800-458-8276
Fax 617-320-9212
TASKING Europe +31 33 558584
TASKING Japan +81 334 050511
All trademarked names are the property of the respective owners
CIRCLE # 26 ON READER SERVICE CARD

sine waves represent the solution to the differential equations describing a system. The principle of superposition that holds for any linear system says we can mix the solutions to get a general solution. Take a vibrating string, for example. Typically, the general solution of that problem results in a variety of sine waves, all rippling back and forth together without interacting. The system behaves as though each sine wave were doing its thing independently of the others. Now, it might seem that a vibrating string is a poor example since we all know that the solutions for such a string lead to only certain frequencies, a fundamental and its harmonics. But that's only true for the unforced motion. We could hook up a piezoelectric transducer, for example, and drive the string at any frequency we can dial in on an oscillator. Indeed, the equations of motion of the string don't preclude sine waves of any frequency—only the boundary conditions force the harmonic solutions, and that's typical of a dynamic system.

The second argument is similar. Suppose we have some medium, say a piece of Styrofoam, and we want to study its sound transmission qualities. Better yet, make it a telephone line. Physically, we know that we can hook up a transducer at one end and a microphone at the other and drive the transducer with any frequency we dial in. We can plot the frequency response of this medium by sampling its response at different frequencies. Unless it's a very special medium indeed, we know deep in our hearts that the frequency response curve is going to be continuous, not a series of lines like we see in Figure 1. If the physical system can transmit frequencies of any value, surely the mathematics must support them as well.

The neatest approach for justifying the integral comes from Hamming. The general idea is to approximate an aperiodic function that we don't yet know how to analyze by using a periodic one that we do. We'll see what happens as the approximation gets better. Consider, for example, the unit pulse shown in Figure 2a that has a pulse width of τ. Mathematically:

$$f(t) = 1 \quad -\tau/2 < t < \tau/2$$
$$f(t) = 0 \quad \text{otherwise} \qquad (32)$$

As a first and very rough approximation to our pulse, we'll use our old friend the square wave, appropriately offset as in Figure 2b. This function matches f(t) perfectly, for that one lone pulse, but of course it repeats elsewhere on the time line with a period T equal to 2τ. We'll call this function $f_2(t)$. All in all, it's a pretty bad fit to f(t), but it's a start.

As the next step, let's make the function longer by sliding out the adjacent peaks on the wave, as in Figure 2c. This is not equivalent to just scaling the wave. We're not changing the pulse width, only the time between pulses. The period increases to, say, 4τ. We'll call this function $f_4(t)$. Eventually, we'll end up with a whole set of such functions, each representing better and better approximations to f(t). The phantom pulses demanded by our condition of periodicity go sliding off the page in opposite directions, heading for infinity, leaving our original pulse standing alone in all its pristine glory. Somewhere out there, those pulses still exist—an infinite number of them, in fact. But for practical purposes, they end up so far out they might as well be gone. I'm going to assert that in the limit they are gone, and that limit occurs when the period of our approximating function reaches infinity.

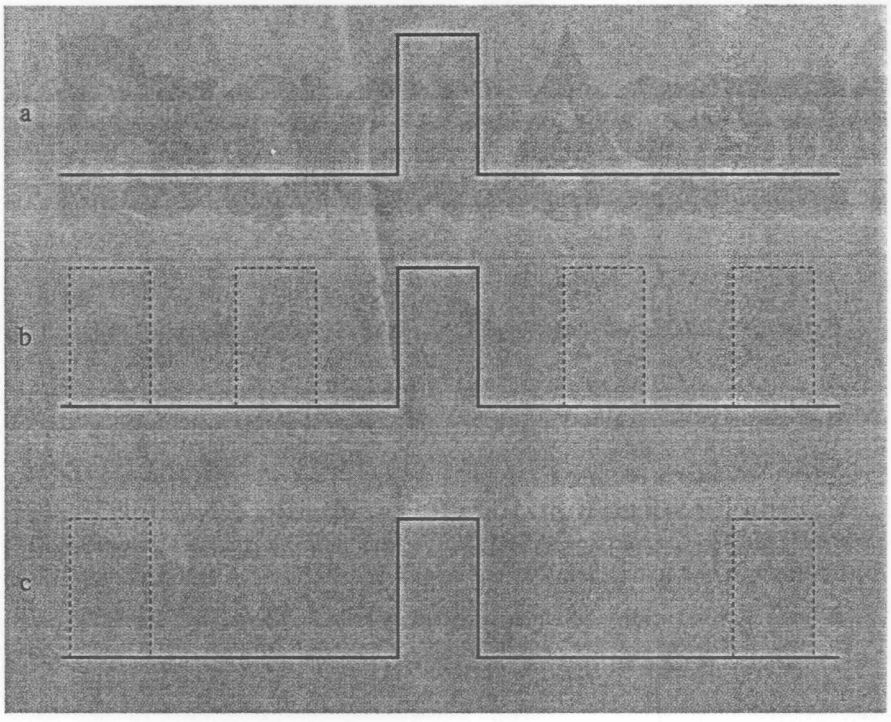
The fourth and final justification is a mathematical one and should be enough to convince even the most hardened skeptic. We'll begin by substituting Equation 30 into Equation 29:

$$f(t) = \frac{1}{T}\sum_{n=-\infty}^{+\infty}\left[\int_{-T/2}^{T/2} f(t)e^{-in\omega t}\,dt\right]e^{in\omega t} \qquad (33)$$



All About Fourier Analysis, Part II

FIGURE 2
Approximations to unit pulse.



Next, let's relabel ω to ω_0, and recall its definition:

$$\omega_0 = \frac{2\pi}{T}$$

Also, we'll define ω to be:

$$\omega = n\omega_0 \qquad (34)$$

The reason for this last step is that we want a variable other than n that changes for the different terms of the summation. ω_0 is a constant (for a given T), but ω is clearly not.

With these definitions, Equation 33 becomes:

$$f(t) = \frac{\omega_0}{2\pi} \sum_{n=-\infty}^{+\infty} \left[\int_{-T/2}^{T/2} f(t) e^{-i\omega t} dt \right] e^{i\omega t} \qquad (35)$$

Next, take a look at the last exponential. This is changing with each increasing value of n because of the increase in ω. And how much does ω increase between successive sums? Why, it's by ω_0 (see Equation 34). We can think of the constant ω_0 as being a $\Delta\omega$, the step between summation points. This is a crucial step because it gives us something approaching an infinitesimal. As T grows larger and larger, $\omega_0 = \Delta\omega$ must necessarily grow smaller. Moving things around a bit gives us:

$$f(t) = \sum_{n=-\infty}^{+\infty} \left[\frac{1}{2\pi} \int_{-T/2}^{T/2} f(t) e^{-i\omega t} dt \right] e^{i\omega t} \Delta\omega \qquad (36)$$

Now we have something that's truly beginning to look like an integral. We've managed to eliminate the integer n everywhere but in the summation itself. Our multiplier looks an awful lot like an infinitesimal at the end. The next step is to consider what happens to the formula as T increases. $\Delta\omega$ must go to zero since it's proportional to $1/T$, giving us two ingredients we need to proceed to the integral: A decreasing infinitesimal, and points that are becoming more closely packed. The only thing that can stop us now is if the range of n somehow gets limited. But that's far from the case because not only is n proportional to ω (see Equation 34), but the proportionality constant is T:

$$n = \frac{\omega}{\omega_0} = \frac{T\omega}{2\pi} \qquad (37)$$

While ω passes through its range, n goes faster and faster to its infinities as T increases. Not only are we still going to get a sum over all values, we now know that the new variable of integration is ω.

In the limit as T approaches infinity, our sum becomes an integral:

$$f(t) = \int_{-\infty}^{\infty} \left[\frac{1}{2\pi} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \right] e^{i\omega t} d\omega \qquad (38)$$

Now we can recognize the two transforms, so we can pull them apart again. We end up with a formal definition for the Fourier transform and its inverse:

$$h(t) = \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} d\omega \qquad (39)$$

$$H(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt \qquad (40)$$

You might have noticed that the place where we put the factor of 2π was pretty arbitrary. We could just have easily have put it in the first integral, and some people do. We could even put it both places (using its square root) to get the ultimate in symmetry. But if we want to talk about frequencies and spectra, it must go where I've shown it.

SERIES OR TRANSFORM?

Like one of those pictures that look like different things depending on how you look at it, we can think of the two integrals in Equations 39 and 40 in two profoundly different ways. On one hand, Equation 39 is merely an extension of the

All About Fourier Analysis, Part II

Fourier series and lets us approximate a function (not necessarily periodic). In that worldview, the companion equation, Equation 40, is just the way we find the coefficient function.

On the other hand, we can think of the equations as defining a transform pair just like the Laplace transform. (In fact, the Fourier transform came first.) Just like the Laplace transform, we can use the Fourier transform in all kinds of interesting ways. In this worldview, the transforms give us the mechanism for moving from the time to the frequency domains. Operating on the assumption that two domains are better than one, and three or four are better than two, this certainly seems a plus. What might be difficult in one domain may turn out to be trivial in another. The more we have access to, the easier life is likely to be.

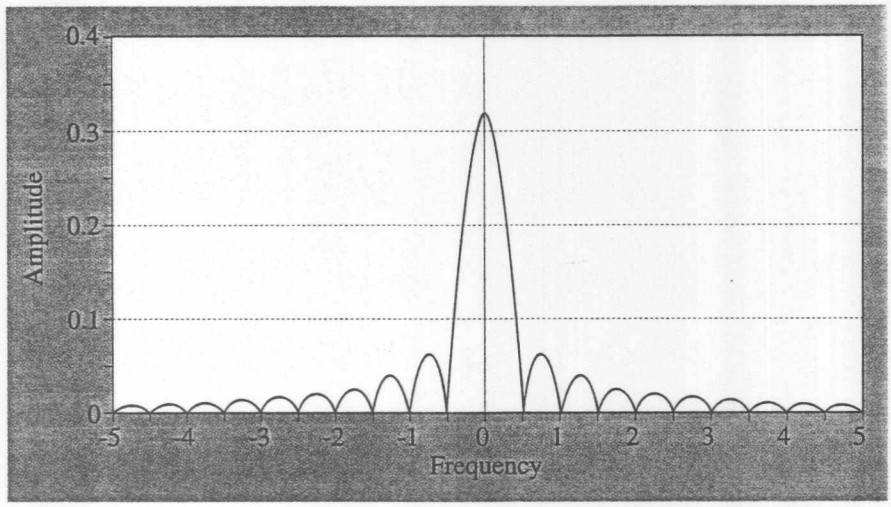
Ironically, now that we've gone to all that trouble to justify the definition of the Fourier integral and Fourier transform using continuous functions, we must now go back the other way because we still can't represent the functions in a computer except as tables of values. In other words, we're going to end up right back where we started, with the routines of Listing 3. But we now know some important things that we didn't know before: The procedures can be used for nonperiodic functions, and the frequency spectrum is really a continuous function, of which we're seeing only a part. We'll continue in that vein next month. Before we leave the world of continuous functions completely, you may be wondering whatever happened to our unit pulse. We now have the tools to compute its transform, so I'll leave that with you as a parting gift.

Recall, the function was defined to be:

$f(t) = 1 \quad -\tau/2 < t < \tau/2$
$f(t) = 0 \quad \text{otherwise}$

Inserting this into our new definition of the transform reduces the range of the integral to be:

FIGURE 3
Transform of a unit pulse.



$$F(\omega) = \frac{1}{2\pi} \int_{-\tau/2}^{\tau/2} e^{-i\omega t} dt \qquad (41)$$

The integral is a fundamental one:

$$\int_{-\tau/2}^{\tau/2} e^{-i\omega t} dt = \frac{1}{-i\omega} \left| e^{-i\omega t} \right|_{-\tau/2}^{\tau/2}$$

$$= \frac{i}{\omega} \left(e^{-\frac{i\omega\tau}{2}} - e^{+\frac{i\omega\tau}{2}} \right)$$

$$= \frac{-i}{\omega} \left(e^{\frac{i\omega\tau}{2}} - e^{-\frac{i\omega\tau}{2}} \right) \qquad (42)$$

But this is almost identical to our definition of the sine function:

$$\sin(x) = \frac{-i}{2}(e^{ix} - e^{-ix})$$

So our result is:

$$\int_{-\tau/2}^{\tau/2} e^{-i\omega t} dt = \frac{2}{\omega} \sin\left(\frac{\omega\tau}{2}\right) \qquad (43)$$

Our transform, then, is:

$$F(\omega) = \frac{1}{\pi\omega} \sin\left(\frac{\omega\tau}{2}\right) \qquad (44)$$

Because of the presence of ω in the denominator, this function has the form of the familiar sine integral:

$$\frac{\sin x}{x} \qquad (45)$$

A graph of the transform is plotted in Figure 3.

That's all we have room for this month. We'll wrap up this series next month with a look at the fast Fourier transform. ESP

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs and holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.

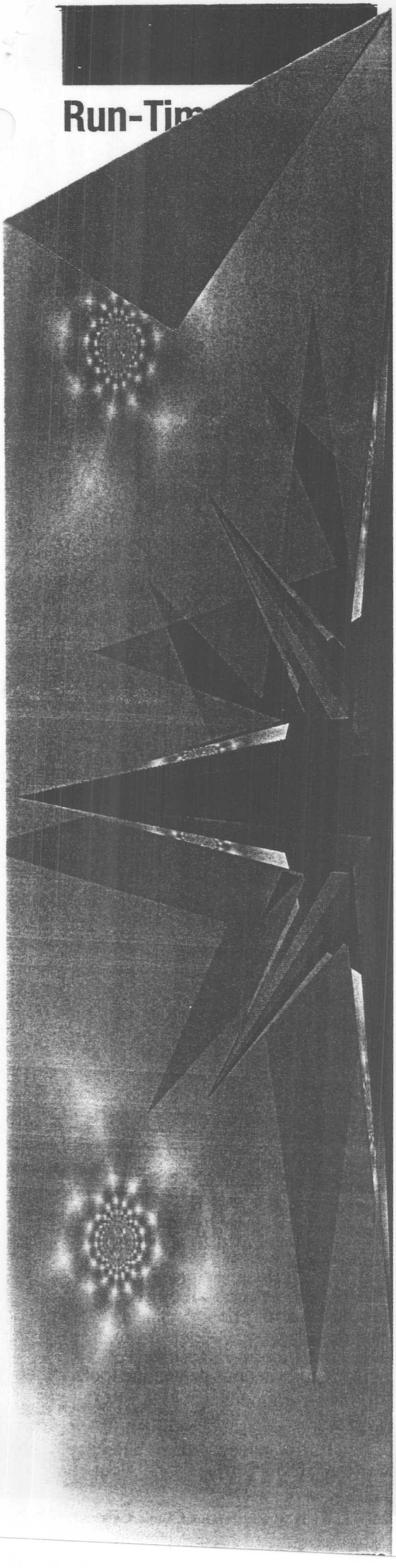
REFERENCES

1. Crenshaw, Jack W., "All About Fourier Analysis, Part 1," *Embedded Systems Programming*, Oct. 1994, pp. 70-90.

2. Crenshaw, Jack W., "The Alphabet from S to Z," *Embedded Systems Programming*, Aug. 1994, pp. 60-73.

3. Plauger, P. J., "Complex Made Simple," *Embedded Systems Programming*, July 1994, pp. 85-88.

4. Hamming, R.W. *Numerical Methods for Scientists and Engineers*. Dover Publications: New York, 1973. pp. 548.



by JACK W. CRENSHAW

All About Fourier Analysis, Part III

The series continues with the theory and software foundation you'll need to understand and implement Fourier transforms.

If you've been following this series, you might have noticed that we have seemed to oscillate between theory and coding. We began with a very mathematical and theoretical treatment of the Fourier series and then swung the opposite way to simple and practical code implementations.

Last month, we continued in the software mode and discussed ways to generalize the previous implementations before returning to a very mathematical treatment, including the motivation for and derivation of the Fourier transform. Here, we'll cover software implementations for the Fourier transform. But first, we'll stick with the pure mathematics and play with the notions of the Fourier transform in its theoretical guise in order to get a good feel for how it works and what it's good for. Next month, we'll wrap the series up with a look at the all-important fast Fourier transform (FFT) and its implementations for both real and complex functions. Hang in there—we're almost done. If everything goes well, you will come out of this series with a set of software implementations in your pocket and, more importantly, the knowledge of how and when to use them.

sub aqua design

At the end of last month's article, I had just introduced the formal definition of the Fourier transform pair:

$$h(t) = \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} d\omega \qquad (1)$$

$$H(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt \qquad (2)$$

The symmetry between these two integrals should be obvious. Except for a sign and the factor of 2π, the forms are identical. In fact, most computer implementations use the same code for both transforms, with only some pre- and post-adjustment for the inverse transform. Once again, this transform represents the logical extension of the Fourier series, in complex form:

$$f(t) = \sum_{n=-\infty}^{+\infty} c_n e^{in\omega t} \qquad (3)$$

where:

$$c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-in\omega t} dt \qquad (4)$$

Remember, the Fourier series only works for functions that are period

with a finite period T. Last month, we saw how the Fourier transform arises as we take the limit as this period goes to infinity, thus allowing us to handle nonperiodic functions.

To get a feel for the way the transform behaves, let's look at a few practical examples. The most simple function of time that comes to mind is a constant:

$$h(t) = 1 \tag{5}$$

You probably know what this function would look like represented as a Fourier series (not transform). All coefficients vanish except for the one constant term c_0, so the transform of this function is:

$$c_0 = 1 \tag{6}$$

But what does it look like when we apply the rules for the integral transform? Substituting the constant for $h(t)$ into Equation 2 gives us:

$$H(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-i\omega t}\,dt \tag{7}$$

I see two problems with this integral. First of all, for any nonzero value of ω, the integrand is simply a sine wave at that frequency. Now, we all know that a sine wave oscillates about zero, and its average value averaged over a full cycle is zero. And an average is exactly what we have for the finite series coefficients defined by Equation 4. But in this case, we're not exactly averaging, since we aren't dividing by the range. More importantly, our range of integration is far from a period; it's infinite. I could wave my arms and assert that the formula is supposed to mean an infinite number of full periods, but I suspect this argument will leave you as unsatisfied as it does me. After all, that range is different for each frequency. It's extremely difficult to argue that we should use different "values" of infinity for each frequency.

You will come out of this series with a set of software implementations in your pocket and the knowledge of how and when to use them.

The second problem is what happens for $\omega = 0$. At that frequency, we get:

$$H(0) = \frac{1}{2\pi}\int_{-\infty}^{\infty} dt = \infty \tag{8}$$

It would appear that our transform of a simple constant function doesn't yield a nice constant after all, but rather a value of infinity at the origin and zero elsewhere. And even that zero seems a little suspect because of the need to assume that the integral of all sine waves is zero. So our simple little first example is not so simple after all. When we understand what's going on for this case, we'll have a much better feel for what the Fourier transform is all about.

IT'S ALL IN THE UNITS

We can get a first hint as to the meaning of our result by looking at the units of each function. Of course, the function $h(t)$ can be used to represent any variable that's a function of time, including voltage, current, distance, velocity, or anything. In the case of the Fourier series, the sine-wave multipliers of each coefficient are dimensionless, so whatever units $f(t)$ has, all the coeffi-

All About Fourier Analysis, Part III

cients c_n have the same units.

But look at the integral in Equation 2. The factor $e^{i\omega t}$ is dimensionless, but our function $h(t)$ is multiplied by a time variable dt. Thus, whatever units $h(t)$ has, the units of $H(\omega)$ cannot possibly be the same. If $h(t)$ is measured in volts, $H(\omega)$ must have units of volt-seconds. Or, since the units of ω are time^{-1}, we can think of $H(\omega)$ as representing volts per unit frequency. (Here, I'm using the term "frequency" a little loosely. Strictly speaking, ω is not frequency but 2π times the frequency. In casual conversation, we tend to use the two terms interchangeably.)

$H(\omega)$ represents a density function—the amount of energy contained in the signal having a frequency between ω and $\omega+d\omega$. For most signals, certainly those that are truly nonperiodic, this amount of energy will be proportional to $d\omega$. But that assumption breaks down when we have a truly periodic signal. In that case, all the energy occurs precisely at specific frequencies. The more we try to narrow down the precise frequency by reducing $d\omega$, the greater the density appears to be. In the limit, as $d\omega$ goes to zero, the density is indeed infinite.

We get a similar result if we compare the graphs of the transform for a periodic and nonperiodic function. Last month, I showed the graph of the Fourier coefficients for a square wave, reproduced here as Figure 1. The square wave is periodic, so we must use the Fourier series. We get a collection of spikes at specific frequencies, as you can see. (In reality, the coefficients of even and odd frequencies alternate in sign but we usually plot only the absolute value of the coefficients since, in general, they may be complex.)

Last month, we also generated the Fourier transform (using the integral) for the nonperiodic function representing a single unit pulse. We found this transform to go like the sine integral:

$$\frac{\sin x}{x}$$

the absolute value of which is reproduced in Figure 2. Comparing the two graphs, you can see that they have a lot in common. In both cases, the amplitude changes as $1/\omega$. But there is also a profound difference. In Figure 1, we have clean spikes of finite height but zero width. In Figure 2, the spikes are smeared out into a spectrum. Each lobe of Figure 2 has not only a finite height but also a finite integral, as represented by the area under the curve. But the spikes of Figure 1 have no "area under the curve" at all—their integrals would be zero since they have no width. This just serves to emphasize that the graphs, though they might look the same, are really plots of totally different things. Figure 1 is a graph of amplitudes, while Figure 2 is a graph of fre-

FIGURE 1
Spectrum for square wave.

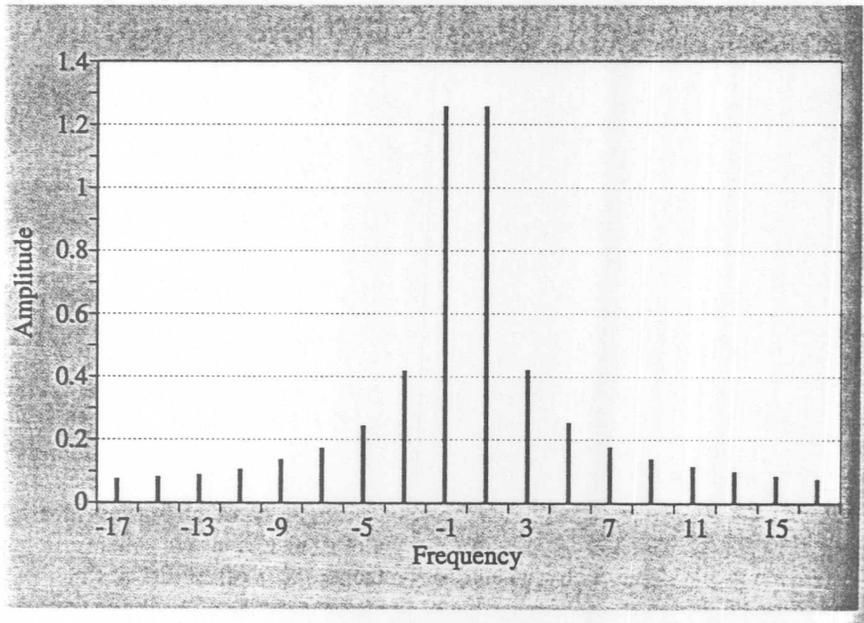
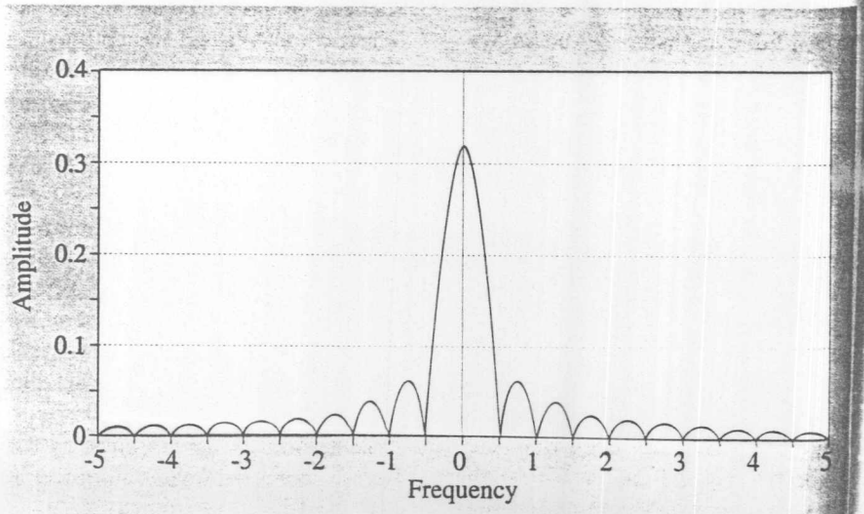


FIGURE 2
Transform of unit pulse.



quency density. Since the spikes of Figure 1 are infinitely dense, we should not be surprised to see them go off the scale when plotted the same way as in Figure 2.

THE DIRAC DELTA FUNCTION

All of this will become clearer with the introduction of the Dirac delta function, always written $\delta(x)$. This peculiar function is zero when x is nonzero and infinite when x is zero. But it is infinite in a special way, so that its integral is one:

$$\int \delta(x)\, dx = 1 \qquad (9)$$

You might have noticed that I didn't show any limits on the integral. That's because it doesn't matter what they are as long as the range includes the value of zero. The integral over all numbers is no different than, say, the integral from -0.001 to +0.001.

Because this strange function is nonzero only for one specific value of x, it serves as a filter or selector for anything else in the integral. Thus:

$$\int \delta(x) f(x)\, dx = f(0) \qquad (10)$$

In controls jargon, the delta function is a unit impulse. It occurs over zero time, so it has infinite amplitude but a finite total effect because its integral is finite, in fact unity. Such impulses can't really exist in nature, but we can come very close. For example, when two billiard balls collide, the force imparted on them is very large for a very short length of time. To compute the motion of the billiard balls, we don't need to know the details of how this force changes over time. The time the balls are in contact is so short that they don't really have time to move much. For all practical purposes, the change in velocity (and spin, if you're a pool shark) of each ball occurs instantaneously. The unit impulse is a mythical but useful idealization of real-world events.

Mathematically, the delta function can be thought of as the derivative of the unit step shown in Figure 2. In the unit step, the input (force, voltage, or whatever) is zero for all time up to the point $t = 0$. At that time, it changes by one unit, whatever the units of the input are. Clearly, the derivative of this input, which is the slope of the curve, is infinite at $t = 0$. And yet, equally clearly, its integral is finite since by definition the integral of a derivative yields the original function. Thus, the unit impulse is precisely the derivative of the unit step.

Let's see what happens when our time function is a delta function or unit impulse. From Equation 2, we have:

$$H(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \delta(t) e^{-i\omega t} dt$$

$$= \frac{1}{2\pi} e^{-i\omega 0}$$

or simply:

$$H(\omega) = \frac{1}{2\pi} \qquad (11)$$

In other words, when our input is the unit impulse, the frequency spectrum is a constant. This is another way of saying that the unit impulse contains all frequencies from D.C. to gamma rays with equal amplitude.

We get an even more interesting result when we perform the inverse transformation. If our function in the frequency domain is a delta function:

$$H(\omega) = \delta(\omega) \qquad (12)$$

(implying, of course, a single frequency of zero), we get:

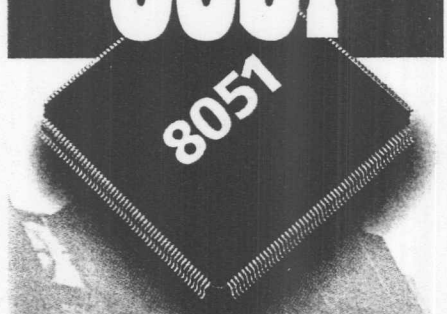
$$h(t) = 1 \qquad (13)$$

which is precisely the constant value we had trouble with going the other way. While a constant value may give us problems in the integration over infinity, the inverse transform is trivial. Fortunately, we have the formula for both transforms, so we can go in whichever direction is the easiest.

What about a single frequency? That

BSO/TASKING

8051



The Best 8051 Software Tools on the Market.

All 8051 manufacturers supported

Highly optimized C and PL/M compilers with assembler, linker and locator

CrossView® Windows debugger with Target ROM or Simulator

Complete ANSI C and runtime libraries including efficient floating point support

Available for DOS and Unix

Small high performance RTOS

Context sensitive help and on-line user manuals

EDE gives you access to all tools through a common interface

Toll-free hot line support

BOSTON SYSTEMS OFFICE TASKING

Discover how fast your application can be.

Ask for your free demo disk and our competitive benchmarks

1-800-458-8276

Fax 617-320-9212

TASKING Europe +31 33 558584

TASKING Japan +81 334 050511

All trademarked names are the property of the respective owners

CIRCLE # 47 ON READER SERVICE CARD

All About Fourier Analysis, Part III

one's easy too, because we can translate the delta function to any point other than the origin by a simple change of variables. There's only one small problem: In the frequency domain, if we have a spike at some value, say Ω, we must have a corresponding one at the negative frequency as well. So our function must look like this:

$$H(\omega) = \frac{1}{2}[\delta(\omega - \Omega) + \delta(\omega + \Omega)] \tag{14}$$

Taking the inverse transform gives us the following:

$$h(t) = \frac{1}{2}\int_{-\infty}^{\infty}[\delta(\omega - \Omega) + \delta(\omega + \Omega)]e^{i\omega t}d\omega$$

$$= \frac{1}{2}[e^{i\Omega t} + e^{-i\Omega t}]$$

so:

$$h(t) = \cos \Omega t \tag{15}$$

(You might be wondering why we got a pure cosine wave with no sine wave or phase shift. That's because of our choice of multiplying constant, unity. In real applications, we'd multiply the amplitude by a constant which might be complex, leading to an arbitrary phase angle.)

WHY TRANSFORM AT ALL?

At this point, you may feel that you've learned more than you care to know about Fourier integrals. But one thing we still haven't discussed to any degree is a very fundamental question, what the heck are they good for? How do they help me get my problem solved?

A hint of the answer comes from considering how other transforms are used. We discussed the use of Laplace transforms in my article, "The Alphabet from S to Z." (Aug. 1994, pp. 60-73.) The idea goes like this: Suppose we have a dynamic system described by a linear differential equation:

$$a_n \frac{d^n x}{dt^n} + a_{n-1}\frac{d^{n-1}x}{dt^{n-1}} + \cdots a_1 \frac{dx}{dt} + a_0 x = u(t) \tag{16}$$

where $u(t)$ is a known but arbitrary forcing function. We'd like to know the behavior of x as a function of time. The key word in the previous sentence is linear, and that point bears emphasis. Everything we've done here, including the Laplace and z-transforms, is based on the assumption that the differential equation is linear, meaning that x and its derivatives enter only as additive terms. For linear equations, the principle of superposition applies; if a func-



Unleash the Power of PromICE

- Connect via ROM Socket; DIP; PLCC; SMT
- Emulate ROMs up to 16MBit in Size
- Fastest Downloads Available: Parallel; Serial; Ethernet
- Run Industry Standard Debuggers
- Target Processor Independent
- Support 3Volt Targets
- Host Software Sources Included
- Shielded Cables for Reliable Operation
- 30-day Money-Back Guarantee; 1Year Warranty!
- Unlimited Phone Support; 24hr BBS

Call Today 1-800-PROMICE (1-800-776-6423)

Grammar Engine Inc.
921 Eastwind Dr., Suite 122 • Westerville, OH 43081
614/899-7878 • Fax 614/899-7888

CIRCLE # 42 ON READER SERVICE CARD

DEVICE PROGRAMMERS



*PC based **$139.95**, Parallel port connect **$449.95**,
Stand alone models **$550.00** or **$750.00** gang
*Read, Pgm and verify 2716 - 8 Megabit EPROM, all models
*Support for micros, Flash, EPROM, 16 bit, PALs, Mach
(call for specific models support list
or download demos via BBS or internet)
*30 day money back guarantee, **FREE** software updates
*Technical support by phone Mon - Fri 8am to 5pm PST
***Made in the USA,** 1 or 2 year warranty on all parts and labor
*Easy to use menu driven software, on line "help" full screen editor
*MACROS, Read and save to disk, Split and set options
***Internet** ftp://ftp.crl.com/users/ro/needhams

FOR MORE INFORMATION CALL

NEEDHAM'S ELECTRONICS, INC.

4630 Beloit Drive #20
Sacramento, CA 95838
(Monday-Friday, 8 am-5 pm PST)

VISA MasterCard C.O.D.

(916) 924-8037
BBS (916) 924-8094
FAX (916) 924-8065

CIRCLE # 43 ON READER SERVICE CARD

tion $x(t)$ is a solution to the homogeneous equation (where $u(t)$ is set to zero), then $2x(t)$, $3x(t)$, or any constant times $x(t)$ is also a solution. If any two functions are solutions, so is the sum of those functions.

Transforms like the Fourier transform and Laplace transform are useful to us only when the differential equation is linear. In fact, for nonlinear systems, terms such as "frequency response" do not even have meaning, since the output of a system driven by a sine wave is not likely to be another sine wave or even the same frequency. Fortunately, most systems in nature are either linear or can be made to appear as such by keeping the signals small and linearizing around some equilibrium state. So in practice, the assumption of a linear system is not a restrictive one. That's quite a lucky thing, too, since we have almost no mathematical tools to deal with the nonlinear system.

Anyhow, when we studied Laplace transforms, we learned that a primary use is to solve dynamic systems for the state function $x(t)$. Taking the transform of Equation 16 gives us:

$$P(s)\,X(s) = U(s) \tag{17}$$

where $P(s)$ is some polynomial in s obtained by substituting s for each derivative in the equation. We called the inverse of $P(s)$ the transfer function $H(s)$ of the system and wrote:

$$X(s) = H(s)\,U(s) \tag{18}$$

Fortunately, a rich set of known Laplace transforms has been tabulated, so controls engineers can usually generate $U(s)$ for most forcing functions of interest. Once the product $H(s)\,U(s)$ has been written down, we can take the inverse transform to obtain $x(t)$.

In "The Alphabet from S to Z," we also learned how to obtain things like the transient response or the frequency response of a system from its transfer function. In particular, the frequency response of a continuous system is obtained by substituting:

$$s = i\omega \tag{19}$$

into the transfer function. Similarly, we get the frequency response of a digital system by substituting:

$$z = e^{ih\omega} \tag{20}$$

So can we use the Fourier transform to do similar things? Yes and no. Of course we can write down the Fourier transforms for differential equations, because we can define the transform of a derivative. By definition:

$$T[\dot{f}(t)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} \dot{f}(t)e^{-i\omega t}dt \tag{21}$$

We can integrate this equation by parts, using the standard formula:

$$\int u\,dv = uv - \int v\,du$$

Let:

$$u = e^{-i\omega t} \tag{22}$$

and:

$$dv = f(t)dt \tag{23}$$

Then:

$$du = -i\omega e^{-i\omega t} \tag{24}$$

and:

$$v = f(t) \tag{25}$$

Our integral becomes:

$$\int_{-\infty}^{\infty} \dot{f}(t)e^{-i\omega t}dt = \left|e^{-i\omega t} f(t)\right|_{-\infty}^{\infty} + i\omega \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

The first term, where the function is evaluated at the infinite limits, must vanish. This is required by the fact that the total integral is finite. If the values

BSO/TASKING

68HC08



Migrate to Performance with the Real HC08 Solution.

Optimized ANSI C compiler supports three memory models

Powerful Motorola compatible macro assembler, linker and locator

CrossView® Windows debugger with Simulator or Emulator environment

Available for DOS and Unix

Complete ANSI C and run-time support libraries

Context sensitive help and on-line user manuals

EDE gives you access to all tools through a common interface

Toll-free hot line support

BOSTON SYSTEMS OFFICE TASKING

Put us to the test.

Ask for your free demo disk and our competitive benchmarks

1-800-458-8276

Fax 617-320-9212

TASKING Europe +31 33 558584

TASKING Japan +81 334 050511

All trademarked names are the property of the respective owners

CIRCLE # 48 ON READER SERVICE CARD

All About Fourier Analysis, Part III

at infinity were anything but zero, this would not be true. So we're left with:

$$T[\dot{f}(t)] = \frac{i\omega}{2\pi}\int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

$$= i\omega\, T[f(t)] \qquad (26)$$

In other words, taking the transform of the derivative of a function is equivalent to simply multiplying the transform of the function itself by $i\omega$. Again, this is very much like the Laplace transform, where the derivative of $x(t)$ transforms to $sX(s)$.

Theoretically, we can deal with differential equations using Fourier transforms but in practice, we almost never do because it's simply easier to use Laplace transforms. In contrast to the rich set of known Laplace transforms, the known Fourier transforms are limited to those we've already discussed: the transforms of sine waves, delta functions, constants, and unit pulses. Also, the Fourier transform has an awkward and difficult condition built in: Its integral ranges for all values of t, from negative infinity to positive infinity. In the real world, I can't possibly know the function over such a range, nor do I want to know it. Worse yet, the inverse transform of a function yields a time function that extends to both positive and negative time. That's a very unsatisfying behavior.

We like to think that actions are reactions are causal. That is, a system responds to some forcing function by moving in a certain way for all time *after* the forcing function is applied. But the fact that the inverse transform extends to negative times implies that the system will have responded to the forcing function by moving before the force was applied. That sort of behavior just seems nonsensical to us.

It was precisely the efforts to fix this problem that led to the definition of Laplace transforms. We don't have time or space to go into the details here, but the general idea is to multiply some function extending over all time by a unit step, which is zero for negative time. That gives us a function extending only over positive time, but the resulting function, as it turns out, does not have a Fourier transform. However, we can approximate the step function by a function representing an exponential decay, which does have a Fourier transform. We can then let the time constant of the exponential function go to infinity, so that it looks like a unit step. The resulting formulas lead directly to the one-sided Laplace transform. In short, the Laplace transform is a direct result of tweaking the Fourier transform to deal with causal systems. Since the Laplace transform is also easier to work with (more known transforms), we really have no incentive to use Fourier transforms for solving differential equations.

So what good are they, then? You already know the answer—we discussed it in the first article of this series. You apply, in effect, a Fourier transform every time you tweak the tone control knobs on your stereo. Remember, Fourier transforms take us from the time domain to the frequency domain and back. In this domain, frequency response is king. Whenever we can describe a system by its frequency response, the frequency domain is the logical domain to operate in. And whenever a signal's frequency content is altered by passing it through a frequency-dependent filter, we can deduce the nature of the altered signal by transforming it to the frequency domain and back again.

The situation is not so much parallel with that of Laplace transforms as orthogonal to it. Recall that, with Laplace transforms, we usually don't care about the detailed time history of our solution; we're only interested in the nature of the system, that is, its sta-

Byte Craft

C Compilers of choice —
for the chip of your choice!

- Fast, efficient optimizing compilers
- Chip specific
- Built-in assembler
- Integrated Development Environment
- Linker, librarian

We respond to your C compiler needs.

BYTE CRAFT LIMITED
Byte Craft Limited
421 King St. N., Waterloo, Ontario
CANADA N2J 4E4
(519) 888-6911
Fax: (519) 746-6751
BBS: (519) 888-7626

MPC for PIC16/17Cxx
C38 for MELPS 740
COP8C for COP8
C6808 for MC68HC08
Z8C for Z8
C6805 for MC6805

CIRCLE # 49 ON READER SERVICE CARD

bility, transient response, frequency response, and so on. While we can theoretically solve for the detailed motion by taking the inverse transform, we're rarely motivated to do so. Laplace transforms are about studying the behavior of dynamic systems.

In contrast, Fourier transforms aren't very useful for studying the behavior of systems, but they're great for studying and reproducing signals that have been filtered in some way. To do that, we typically need both the forward and inverse transforms.

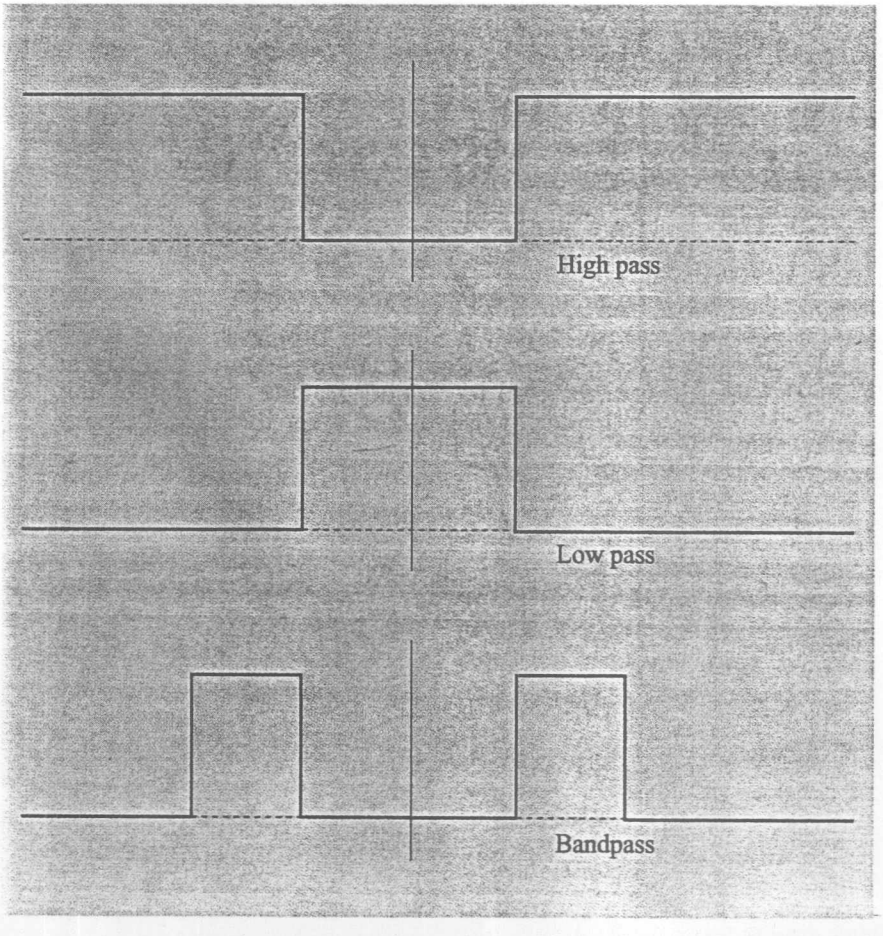
Each transform has its uses. Conceptually, we can describe a given function in any one of the available domains:

$$h(t) <=> H(s) <=> H(z) <=> H(\omega) \quad (27)$$

Each domain has its strengths and weaknesses. The time domain is the one we live in, of course, so ultimately we'd like our results to end up here. Also, things like data sampling, gating, and delays are best described in this domain. The s-domain is great for studying stability and designing control systems. The discrete-time world of z is essential if we're dealing with digital systems. Not surprisingly, frequency-dependent effects are best described in the ω-domain of Fourier transforms. Three ideal filters, high-pass, low-pass, and bandpass, are shown in Figure 3. The description of these filters is trivial in ω-space—they're the simple rectangles shown in the figure. Describing these ideal filters in any other domain is anything but trivial.

The bottom line is this: The more domains we have to work with, and the

FIGURE 3
Ideal filters.



BSO/TASKING
80C196
intel
The New Standard in C196 Tools.
Highly optimized C compiler with Intel compatible assembler, linker and locator
ANSI C and run-time libraries supplied including floating point support
ChipView® debugger with Target ROM environment
Small high performance RTOS
On-line user manuals with context sensitive help
EDE gives you access to all tools through a common interface
Support hot line only a toll-free call away
BOSTON SYSTEMS OFFICE
TASKING
Put the standard to work for you.
Ask for your free demo disk and our competitive benchmarks
1-800-458-8276
Fax 617-320-9212
TASKING Europe +31 33 558584
TASKING Japan +81 334 050511
All trademarked names are the property of the respective owners
CIRCLE # 50 ON READER SERVICE CARD

All About Fourier Analysis, Part III

more tools we have to work in them, the easier our work will be. Everything that we do in the s, z, or frequency domains can be done only in the time domain; the job just becomes that much harder. The Fourier transform is one more tool for one more domain to make our lives easier. We use it when it's useful to us. We don't use it to determine frequency responses of systems because the Laplace and z transforms do that task better. Where the Fourier transform shines is in the tweaking of the frequency content of a signal via filters.

THE CONVOLUTION INTEGRAL

Until now, I've studiously avoided dealing with the notion of a convolution integral, mainly because we haven't had the tools to talk about it. Now that we have these tools, it's time to introduce the idea. It depends on two key concepts that you should by now have firmly in hand:

- The principle of superposition
- The unit impulse.

Suppose we'd like to pass a signal $x(t)$ through a black box, and record the output signal $y(t)$. We don't know yet what the response of the black box will be, but we do know how it responds to a unit impulse. Now suppose we tested it with a unit impulse and found it to have a response $g(t)$, which we'd then use to deduce the output $y(t)$.

The key to the solution is the principle of superposition. If the system inside the box can be described by a linear system, we're safe in assuming that we can treat the input signal as a superposition of signals. In particular, we can imagine it to be made up of an infinite number of tiny signals, each of which is an impulse.

We already know that the response to an impulse at $t = 0$ is $g(t)$, and that the response to an impulse at any other time τ is $g(t-\tau)$. For the impulse corresponding to time τ, its height is $x(\tau)$, making the response of the system:

$$x(\tau)\, g(t-\tau)$$

The overall response is represented by the sum of all these little impulse responses, or:

$$y(t) = \int_{-\infty}^{\infty} x(\tau)g(t-\tau)\,d\tau \qquad (28)$$

I've extended the integral over all time, partly because we'd like to have results that work for noncausal as well as causal systems and partly because that's the range the Fourier transform

MS-DOS®

Licenses for Embedded Systems

Now you can choose any Microsoft MS-DOS version for your system, including earlier, more compact versions.

Annabooks is now issuing large and small-quantity production licenses for all Microsoft MS-DOS **versions from 3.3 to 6.22**. All are ROMable, using Annabooks' PromKit utility. Windows and Windows NT are also available.

Also, we have versions of MS-DOS that actually **execute in ROM**, using only 19K of RAM. ROMable Windows versions too!

More from Annabooks for Embedded System Designers!

- **PromKit:** Put MS-DOS or other code in EPROM as a read-only disk. Includes source code!
- **DOS Buttons:** Create pull-down menus, on-screen calculators – all without the Windows overhead!
- **BiosMaker:** Do your own! Complete BIOS files & source code, instructions, debug routine.
- **RTKernel:** A real-time, multitasking executive that runs under MS-DOS!
- **Design Publications:** More about embedded systems than you can find anywhere else. Also courses on Embedding MS-DOS, PCI, ISA/EISA, PCMCIA, Fuzzy Logic, others.

Call Toll Free **(800) 462-1042**
We accept Co. PO's and
Amer. Express, Visa, MasterCard

AB Annabooks
11848 Bernardo Plaza Court
San Diego, CA 92128-2417
(619) 673-0870 • Fax: (619) 673-1432

CIRCLE # 51 ON READER SERVICE CARD

Protected Mode Stress?



Ease into Protected Mode with pmEasy

pmEasy is a complete protected mode environment for embedded systems. It initiates protected mode and provides an application loader, trap handler, error handler, memory manager, debugger support, screen writes and more. *pmEasy* is integrated with low-cost 16- and 32-bit development tools from Microsoft, Borland, and others.

Why struggle developing your own pm environment? *pmEasy* is proven code which lets you focus on your application.

*pmEasy*16 or 32 **$495** including source code

BUY AND TRY 30 day money-back guarantee

MICRO DIGITAL INC 1-800-366-2491
Developer of SMX
Cypress, CA, USA FAX 714-891-2363 **VISA, MC, AMEX**

CIRCLE # 52 ON READER SERVICE CARD

wants anyway. Equation 28 is the convolution integral. If we decide to work purely in the time domain, we must be prepared to evaluate this integral for each input signal. Now let's take the Fourier transform of this integral:

$$Y(\omega)=\frac{1}{2\pi}\int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty}x(\tau)g(t-\tau)d\tau\right]e^{-i\omega t}dt$$

If we exchange the order of integration, this becomes:

$$Y(\omega)=\frac{1}{2\pi}\int_{-\infty}^{\infty}x(\tau)\left[\int_{-\infty}^{\infty}g(t-\tau)e^{-i\omega t}dt\right]d\tau \tag{29}$$

For the inner integral, let:

$u = t - \tau$

so that:

$t = u + \tau$

and:

$dt = du$

Then the inner integral becomes:

$$\int_{-\infty}^{\infty}g(u)e^{-i\omega(u+\tau)}du$$

$$= e^{-i\omega\tau}\int_{-\infty}^{\infty}g(u)e^{-i\omega(u+\tau)}du$$

$$= 2\pi\, e^{-i\omega\tau}G(\omega) \tag{30}$$

Equation 29 then becomes:

$$Y(\omega) = 2\pi\, G(\omega)\left[\frac{1}{2\pi}\int_{-\infty}^{\infty}x(\tau)e^{-i\omega\tau}\,d\tau\right]$$

or:

$$Y(\omega) = 2\pi\; G(\omega)X(\omega) \tag{31}$$

Except for that bothersome factor of 2π that most authors manage to ignore, the resulting transform is simply the product of the other two transforms. So we can get the response of the system to any input by multiplying two tranforms together and then performing an inverse transform on the product.

We get precisely the same kind of result (but without the 2π) when we're dealing with Laplace transforms and z-transforms. If you agree that multiplying two functions that are typically ratios of polynomials is easier than integrating a convolution integral, then you understand the value of integral transforms. It's precisely this: We can compute the responses of very complex systems to very complex signals simply by multiplying the individual transfer functions.

The process of computing the response of a system to a given signal goes like this:

- Compute the transform of the signal.
- Compute the transfer function for the system, multiplying individual transfer functions if necessary.
- Multiply the signal and the system transforms.
- Take the inverse transform of the result.

This process applies to all transforms. The two distinguishing differences for the Fourier transform are that the frequency response is the transfer function, since we're in the frequency domain, and that we typically find the frequency response via some other mechanism, except when we're dealing with filters like those of Figure 3.

Depending on the nature of our system, we may find that certain parts are best done in the frequency domain and other domains. I read of one simulation that required eighteen separate transformations to deal with different parts of the problem. In cases such as this, we need to be able to compute the transforms very efficiently. Unlike the Laplace transform, in which we can often get the desired transform from a table, we're almost certain to have to



AMX™
The Real-Time Multitasking Kernel

680x0, 683xx
80x86/88 real mode
80386 protected mode
i960® family
R3000, LR330x0
Z80, HD64180

Features

NEW
• 29K™ support
• File System
• TCP/IP

- Full-featured, compact ROMable kernel with fast interrupt response
- Preemptive, priority based task scheduler with optional time slicing
- Mailbox, semaphore, resource, event, list, buffer and memory managers
- Configuration Builder utility eases system construction
- InSight™ Debug Tool is available to view system internals and gather task execution statistics
- Supports inexpensive PC-hosted development tools
- Comprehensive, crystal clear documentation
- No-hidden-charges site license
- Source code included
- Reliability field-proven since 1980

Count on KADAK.
Setting real-time standards since 1978.

For a ***free*** *Demo Disk and your copy of our excellent AMX product description, contact us today.*

Phone: **(604) 734-2796**
Fax: **(604) 734-8114**

KADAK Products Ltd.
206 - 1847 West Broadway
Vancouver, BC, Canada V6J 1Y5

AMX is a trademark of KADAK Products Ltd.
All trademarked names are the property of their respective owners.

CIRCLE # 54 ON READER SERVICE CARD

do it numerically in the case of the Fourier transform. Hence the intense interest in the FFT.

BACK TO THE REAL WORLD

Let's leave the world of theory now and get back to practical implementations. The transition may surprise you. Ironically, after all our efforts to extend the notion of the Fourier series, with its discrete coefficients to the infinite and integral world of the Fourier transforms, we're now heading back the other way. The reason is this: We can't store an infinite, continuous function in the computer, only a finite set of samples. So the integrals we worked so hard to derive must become summations again. The routines I've given you for the Fourier series work just dandy for Fourier transforms; we need only change the interpretation of the results.

Take a look at the forward Fourier transform:

$$F(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \qquad (32)$$

If we're going to represent the function $f(t)$ as a table of numbers, our first challenge is to deal with the range, particularly the "minus infinity" side of the integral. For practical reasons, we'd like to imagine the function beginning at $t = 0$ and moving forward, but that's not what the integral calls for, is it?

I have two arguments to give that allow us to treat the function as beginning with time zero. All three arguments are pretty weak, and strike me more as arm-waving than fact, but they're all we have. The first involves going back to the Fourier series, where the integral had a finite range:

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{+\pi} f(x)e^{-inx}dx$$

The important thing to notice is that the two end points, $-\pi$ and π, are really the same point: The angle is the same either way, and so is the function of that angle. We're not so much integrating over a range as we are integrating around a circle. For that reason, it really doesn't matter much where we break the circle, and changing the range to $0..2\pi$ gives an equally valid (but different) representation. In fact, using any starting point is equally valid.

In the case of the Fourier transform, the mathematics says to integrate over all possible numbers, but once we make the commitment (as we must) to chop the function off at some point and make it finite at both ends, it really doesn't matter where we make the chops.

The second argument is related to the first and is based on the recognition that any finite representation of the function takes us right back from a nonperiodic to a periodic function. The idea is that we'll consider a tabular function to be not the whole function of time, but only typical of it, and we'll mentally extend the function over all time by tacking copies on at both ends. After all the trouble we went to to define the Fourier transform to deal with nonperiodic functions, it seems a bitter pill to have to recognize that we're back to periodic functions again. But that's the fact of the matter.

This situation is very much like the artifice we used to arrive at the Fourier transform in the first place. Recall that we began by approximating a single pulse as a square wave. Then we imagined stretching the waveform out without changing the shape of the pulse, so that the periodic pulses became farther and farther apart. At the limit, we imagined they'd moved all the way out to infinity, where they'd be no trouble. In fact, they're really not at infinity—they're still hovering out there on the horizon. The best we can hope for is to make the distance between them far enough that the results we get will approximate the integral transform. And the notion of approximation is the bottom line. The instant we decide to represent an infinitely long, continuous function by a finite table of points,

we're no longer talking about an exact solution but an approximation to it, and we can only hope that the approximation is good enough.

If you can accept this admittedly arm-waving argument, then we can represent $f(t)$ by a tabular function containing N points. We'll call this function h_k and define the relationship between k and t by:

$$t_k = kh \tag{33}$$

where h is the sampling interval. Equation 33 then becomes:

$$F(\omega) = \frac{h}{2\pi}\sum_{0}^{N-1} f_k e^{-i\omega kh} \tag{34}$$

Take a look at that coefficient in front. What is its relationship to N, the number of points? Well, if we accept the fact that we're really working with a periodic series again, we must also accept the fact that our new range is $0..2\pi$ radians of angle, and that the sampling interval h breaks this range up into N parts.

Let T be the total time span covered by our tabular function. That is:

$$T = Nh \tag{35}$$

This, of course, is the period of our newly periodic function. Now, where there's a period, there's a frequency given by the reciprocal of that period and a base angular rate:

$$\Omega\pi = 2\pi/T \tag{36}$$

Remember, we're back to periodic functions, so the frequency spectrum is no longer continuous but discrete. We shouldn't have a problem with that concept at this point, because we can only represent a continuous function by a table. As when we were defining the Fourier transform, we can think of Ω as being a $\Delta\omega$, or the difference between two successive frequencies in the table.

Eliminating T between Equations 35 and 36 gives us a relationship between h and N:

$$\Omega = \frac{2\pi}{Nh} \tag{37}$$

or:

$$h = \frac{2\pi}{\Omega N} \tag{38}$$

or:

$$\frac{h}{2\pi} = \frac{1}{\Omega N} \tag{39}$$

This is the coefficient in Equation 34, which becomes:

$$F(\omega) = \frac{1}{\Omega N}\sum_{0}^{N-1} f_k e^{-i\omega\, kh} \tag{40}$$

We've now got the first half of the transition from continuous to discrete functions. We get the second half by repeating the process with ω. We already have a big hint as to how that should go from the notion that Ω is equivalent to a $\Delta\omega$. We can define a set of discrete values of ω given by:

$$\omega = n\Omega \tag{41}$$

If we substitute this definition into Equation 39, it gives us our final transform:

$$F_n = \frac{1}{\Omega N}\sum_{0}^{N-1} f_k e^{-ink\Omega h} \tag{42}$$

This is the official, rigorous definition of the discrete Fourier transform, and is valid for both real and complex functions f_k.

A word of warning: You will not find this equation in precisely this form in any textbook, at least none of the books I've researched. That's because people tend to play a little fast and loose with minor details of the definitions. I frankly admit that what was most difficult for me in preparing this series was resolving seemingly misplaced factors of 2π and similar details between one textbook and another. Few authors even define the trans-

BSO/TASKING

R3000



No Risk.
Fast Applications.
No Competition.

The most efficient software development solution available for R3000 applications

Optimizing ANSI C compiler, assembler and incremental linker

ANSI C, run-time and floating point libraries

Multi-tasking locator

CrossView® Target ROM debugger

EDE gives you access to all tools through a common interface

CrossView® and our Board Support Packages provide complete integrated solutions for all LSI LOGIC and IDT R3000 evaluation boards

BOSTON SYSTEMS OFFICE TASKING

No argument, the best is BSO/TASKING.

Ask for your free demo disk and our competitive benchmarks
1-800-458-8276
Fax 617-320-9212

TASKING Europe +31 33 558584
TASKING Japan +81 334 050511

All trademarked names are the property of the respective owners

CIRCLE # 55 ON READER SERVICE CARD

All About Fourier Analysis, Part III

forms the same way, so don't be surprised if you find differences between what you see here and what you see elsewhere. I don't claim to be a more authoritative source than any other reference, but I have made considerable effort to keep things consistent.

In particular, you will almost always find the formula for the series in Equation 42 with the Ω missing. Not only can this be confusing, it isn't even dimensionally correct. Remember, unlike the case with the Fourier series, the Fourier transform and its inverse do not have the same units; they differ by a factor of inverse time, which is precisely the units of Ω. Folks get away with this by normalizing the coefficients to an angular rate of one, corresponding to a period T of 2π (in whatever time units you'd like). If we do the same thing here, we'll get (using Equation 38):

$$F_n = \frac{1}{N}\sum_{0}^{N-1} f_k e^{-2\pi\, ink/N} \tag{43}$$

This is the form you're much more likely to see. It also happens to be precisely the same form we got for the coefficients of the complex Fourier series. What does this mean? Simply that if we normalize the angular rate to one, we already have the code for the Fourier transform, at least the transform for real functions. It's the function `fourfit` given last month and repeated here in Listing 1 with a more appropriate name.

One small item that remains to be dealt with is the range of the index n. Remember that in the complex Fourier series, the coefficients c_n range through negative as well as positive indices, with the negative indices corresponding to the negative frequencies. In function `fourfit` and its companion `fourier`, we did not need or allow the negative-frequency coefficients because we limited our attention to real functions. For real functions, the negative-frequency coefficients are the complex conjugates of their positive-frequency counterparts, so we didn't have to store both halves. But what about when we have to deal with negative indices? C and C++ won't let us use negative indices, so how do we shift things around?

LISTING 1
Discrete Fourier transform.

```
// Compute Fourier coefficients
void fourfit(Vector &f, C_Vector &c){
  int n = f.rank();
  int m = c.rank();
  assert(m <= n/2);
  complex w = exp(complex(0, 2.0 *
    pi/n));
  complex wi = complex(1, 0);
  complex wij;
  c = 0;
  for(int i=0; i < n; i++){
    wij = complex(1, 0);
    for(int j=0; j < m; j++){
      c[j] += f[i] * wij;
      wij *= wi;
    }
    wi *= w;
  }
  c /= n;
}
```

Fortunately, a simple solution exists; it's the same one we used to define $f(t)$. We haven't yet talked about a phenomenon called aliasing, but we have talked about Shannon's limit and the fact that you can't get information out of a function at a frequency higher than half the sampling frequency. Aliasing is the reason for that, and we'll learn more about it next month. Briefly, the phenomenon causes false spectra to appear at intervals of the sampling frequency. Usually, this is an unwelcome phenomenon and something to be avoided. In this case, though, it's very welcome because it means that we'll find those missing negative-frequency coefficients repeated again at positive indices above $N/2$. The situation parallels the switch from the range $-\pi..\pi$ to $0..2\pi$. By extending our index range over the full N points, we pick up the aliased version of the negative-frequency coefficients above the $N/2$ limit.

Using this concept, and following a similar procedure for the transition from integral to sum, we get the inverse discrete Fourier transform:

$$f_k = \Omega\sum_{0}^{N-1} F_n e^{ink\Omega h} \tag{44}$$

or, with the same normalized rate:

$$f_k = \sum_{0}^{N-1} F_n e^{-2\pi\, ink/N} \tag{45}$$

We now have forms for both the Fourier transform and its inverse in discrete form. They are:

$$f_k = \Omega\sum_{0}^{N-1} F_n e^{ink\Omega h}$$

$$F_n = \frac{1}{\Omega N}\sum_{0}^{N-1} f_k e^{-ink\Omega h}$$

Before we close, I'd like to show you one final form for both, in which we retain the rigor but in terms of T instead of Ω. In this form, the transform pair is:

$$f_k = \frac{2\pi}{T}\sum_{0}^{N-1} F_n e^{2\pi ink/N} \tag{46}$$

$$F_n = \frac{T}{2\pi N}\sum_{0}^{N-1} f_k e^{-2\pi\, ink/N} \tag{47}$$

Again, these equations reduce to more common forms when we normalize the period so that $T = 2\pi$. ESP

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.